EXHIBIT C

C1

```
1    /*
2        Copyright 1996,1997 EMC Corporation
3    */
4
5    /*
6        EDMmain.c
7
8        Mission Statement: This is the main service file for the EDMsession
9                           daemon.
10                          This file contains the main loop,
                            and all calls required
                            to prepare the daemon to go off and service
                            RPC's.
11
12       Primary Data Acted On:
13
14
15
16       Compile-Time Options:
17
18                          USE_SUNRPC - Compile source with sunrpc
19                                       support.  If
                                         not set, assume DCE support.
20
21                          NONPRODUCTION - Compile source for in house.
                                            developer
                                            testing on local work station
                                            Should
                                            only be used for targeted
                                            testing.
22
23       Basic idea here: Initialize required locks,
24                        register RPC interface, establish signal handlers,
25                        go wait for RPCs.
27    */
28    /*
29       The following provides an RCS id in the binary that can be located
30       with the what(1) utility.  The intent is to keep this short.
31    #if !defined(lint)
32    static char  RCS_id [] = "@(#)$RCSfile$ "
33                             "$Revision$ "
34                             "$Date$ " ;
36    #endif
37    /* #define _POSIX_SOURCE    unable to compile with this define set */
38    /* #define _XOPEN_SOURCE    unable to compile with this define set */
40    /**********************************************************************/
41    /*
43       Routine: main
44
45       Inputs:  argc, argv
46
48       Outputs: None
49
50       Return Codes:
51                exit status
52
53       Purpose: This is the main routine which sets up the daemon
                  to handle RPC calls.
                  and handles them until it is told
                  to stop or it sees a fatal error.
55       Intended caller: None
56
```

```
57    **********************************************************************/
58
59    /*  main  */
60    void
      main (int argc, char *argv[])
61    {
62        /* Parse options */
64        (void) parse_commandline(argc, argv);
65
67        /* Setup logging */
68        (void) daemon_initialize_logging();
69
71        /* Enable permanent interrupt catching */
73        (void) daemon_catch_interrupts();
74
75        /* Function may not return if improper user running daemon */
77        (void) daemon_become_daemon();
79
81        /* Function will not return if this fails */
83        (void) daemon_check_proper_ID();
85
87        /* This function doesn't return on failure */
89        (void) daemon_initialize();
91
92        /* Re-establish log initialization since all "fd's" were
94           closed by esl_daemon_startup (in daemon_become_daemon) */
96        (void) daemon_specific_initialization();
98
100       /* Unregister service, cleanup cache... Never returns */
102       (void) daemon_cleanup();
104
106       /* Strictly to inhibit compiler warning... */
112       return( 0 );
114
115   }
```

```
  1   /*
  2   **      Copyright 1996, 1997 EMC Corporation
  3   **
  4   **
  5   **
  6   **
  7   **
  8   **
  9   **      Mission Statement: EDMRestoreEng.c  This is the main service file for the EDMessd
                                daemon.  This
                                file contains the callbacks from the main
                                function which
                                prepares the daemon to go off and service RPC's.
 10   **
 11   **
 12   **
 13   **
 14   **      Primary Data Acted On:
 15   **
 16   **
 17   **      Compile-Time Options:
 18   **
 19   **              USE_SUNRPC - Compile source with sunrpc
 20   **                           support.  If
                                  not set, assume DCE support.
 22   **
 23   **      Basic idea here: Module for UNIX specific daemon initialization
 24   **
 25   **
 26   **      The following provides an RCS id in the binary that can be located
 27   **      with the what(1) utility.  The intent is to keep this short.
 28   */
 29   #if (defined(lint))
 31   static char     RCS_id [] = "@(#)$RCSfile: EDMRestoreEng.c,v $ "
 32                               "$Revision: 1.23 $ "
 34                               "$Date: 1997/02/06 20:49:15 $ " ;
 35   #endif
 37   /* #define _POSIX_SOURCE       unable to compile with this define set */
 39   /* #define _XOPEN_SOURCE       unable to compile with this define set.  If
 40
 41   #include <stdarg.h>
 42   #include <string.h>
 43   #include <syslog.h>
 44   #include <pthread.h>
 45   #include <thread.h>
 47   #include <sys/utsname.h>
 49   #include <stdlib.h>
 50   #include <stdio.h>
 51   #include <csc/cscomm.h>
 53   #include <restore/csc_EDMRestoreEng.h>
 55   #include <EDMmain.h>
 56   #include <util/esl_core.h>
 57   #include <EDMRestoreEngApi.h>
 58   #include <EDMProcessManager.h>
 59   #include <EDMProgress.h>
 60   #include <EDMErr_scr.h>
 61   #include <EDMErr.h>
 62   #include <EDMRR_ccw.h>
 63   #include <EDMRCommandApi.h>
      #include <EDMRRQuestionApi.h>
      #include <EDMRGratingApi.h>
```

```
 65   **      Need to define _XOPEN_SOURCE for signal function definitions
 66   **      and certain signal structure definitions.
 67   */
 68   #define _XOPEN_SOURCE
 69   #include <signal.h>
 71   #undef _XOPEN_SOURCE
 73   static rpc_if_handle_t if_spec;
 75   static int      G_debug = FALSE;    /* Variable which will disable forking */
 77   static char     **commandlineargs; /* Pointer to command line args */
 79
 81   /*****************************************************************************
 82   **
 83   **      Routine: IsDebugOn
 84   **
 85   **      Inputs: None
 86   **
 87   **      Outputs: None
 88   **
 89   **      Return Codes: TRUE if debug is on.
 90   **
 91   **      Purpose: This routine can be used to tell other subsystems
 92   **               whether debugging is available.
 94   **
 95   **      Intended caller:  internal only.
 96   **
 97   *****************************************************************************/
 98   boolean_ty
 99   IsDebugOn()
100   {
101   #ifdef DEBUG
102 1    return TRUE;            /* if DEBUG defined, we must be in debug mode */
104   #endif
106 1    if (debugmode)          /* if turned on manually via adm, this is it */
107 1        return TRUE;
108 1    return G_debug;         /* default is how we were started: -d menas debug */
109 1 }
```

```
111  *************************************************
112  **
113  ** Routine-Kill handler
114  **
115  ** Inputs:  int signal - the signal which was received.
116  **
117  ** Outputs: Will log messages telling what action is being taken.
118  **
119  ** Return Codes:
120  **     exit with the number of the signal received
121  **
122  ** Purpose: This routine handles specific signals i.e. SIGINT,
123  **          SIGQUIT, SIGTERM. Each results in a log entry and an exit.
124  **
125  ** Intended caller: internal only.
126  **
127  *************************************************
128  */
129  static void kill_handler( int signal )
130  {
131     error_status_t    status;
132     time_t            current_time;
133     char              *ctimebuf;
134     char              *ebuff = NULL;
135
136  1    /* If main exits, it calls this routine with signal 0 */
137
138  1    /* Unregister the interface */
139  1    (void) csc_unregister_server_interface(&if_spec, &status);
140
141  1    /* If the unregister fails, report the problem, but continue */
142  1    if ( status != error_status_ok )
143  2    {
144  2       ebuff = (char *) csc_get_error( status );
145
146  1       (void) EDMRestoreEng_logent(
147  1          __FILE__, __LINE__, LOG_ERR, MESSAGE_NO_LOGIN, 0,
148  1          "CSC_SERVER_LOGIN failed: <%d> %s",
149  1          status, (ebuff ? ebuff : "Unknown error") );
151  1
152  1    /* Get the current time */
154  1    (void) time(&current_time);
156  1    ctimebuf = ctime(&current_time);
157
159  1    /* Overlay newline with null - buf should always be 26 bytes long */
160  1    ctimebuf[ strlen(ctimebuf) - 1 ] = 0;
162
164  1    exit(signal);
     }  /* End of kill_handler() */
```

```
167  *************************************************
168  **
169  ** Routine: unregister_csc
170  **
171  ** Inputs:  none
172  **
173  ** Outputs: Will log messages telling what action is being taken.
174  **
175  ** Return Codes:
176  **     none
177  **
178  ** Purpose: This routine handles the csc_unregister call
179  **          before exit
180  **
181  ** Intended caller: internal and process manager before exit
182  **
183  *************************************************
184  */
185  void unregister_csc( void )
186  {
187     error_status_t    status;
188     char              *ebuff = NULL;
189
190  1    /* Unregister the interface */
191  1    (void) csc_unregister_server_interface(&if_spec, &status);
192
193  1    /* If the unregister fails, report the problem, but continue */
194  1    if ( status != error_status_ok )
195  2    {
196  2       ebuff = (char *) csc_get_error( status );
197  2       (void) EDMRestoreEng_logent(
198  2          __FILE__, __LINE__, LOG_ERR,
199  2          MESSAGE_CANNOT_UNREGISTER, 0,
200  2          "CSC_UNREGISTER_SERVER failed: <%d> %s",
201  1          status, (ebuff ? ebuff : "Unknown error") );
203  1    }
204     return;
     }
```

```
/*************************************************************

 *
 * Function Name:
 *    display_usage
 *
 * Simply displays the usage
 *
 * Call Arguments:
 *    Inputs:
 *       program name
 *
 * Error Outputs and Side Effects:
 *    Prints usage.
 *
 * Special Considerations:
 *    None.
 *
 *************************************************************/
static void
display_usage (IN char *programame)
{
    /* Print out usage smsc. */
    fprintf (stderr, "Usage: %s [-d]\n", programame);
    fprintf (
        stderr,"-d keep the daemon from forking so debugging is easier\n");
} /* end display_usage () */
```

```
/*************************************************************

 *
 *
 * Routine: daemon_catch_interrupts
 *
 * Inputs:
 *    None
 *
 * Outputs:
 *    None
 *
 * Return Codes:
 *    None
 *
 * Purpose:
 *    Sets up signals for service. On NT we will have to
 *    consider what OS constructs to replace signals with.
 *    In this case we are catching SIGTERM, SIGINT, and
 *    SIGQUIT and ignoring anything else.
 *
 * Intended caller: internal only.
 *
 *************************************************************/
void daemon_catch_interrupts()
{
    struct sigaction    sactions;    /* signal actions */

    ZERO( sactions );

    /*
     * Set an empty list so we can set signals we want to handle
     */
    (void) sigemptyset( &sactions.sa_mask );

    /*
     * Add signals that we want to handle
     */
    (void) sigaddset( &sactions.sa_mask, SIGTERM );
    (void) sigaddset( &sactions.sa_mask, SIGINT );
    (void) sigaddset( &sactions.sa_mask, SIGQUIT );

    /* Setup the signal handler */
    sactions.sa_handler = kill_handler;

    /*
     * Assign handler to each signal we are interested in.
     */
    (void) sigaction( SIGTERM, &sactions, NULL );
    (void) sigaction( SIGINT, &sactions, NULL );
    (void) sigaction( SIGQUIT, &sactions, NULL );

    /*
     * Setup mask so we can specify what signals we will ignore.
     */
    (void) sigfillset( &sactions.sa_mask );

    /*
     * We want to ignore everything except those we have set up
     * above so remove those from the list.
     */
    (void) sigdelset( &sactions.sa_mask, SIGTERM );
    (void) sigdelset( &sactions.sa_mask, SIGINT );
    (void) sigdelset( &sactions.sa_mask, SIGQUIT );
```

```
296  1    *  Set the mask.  Since no other threads have been started,
297  1    *  all threads will get this mask.
298       */
299  1    (void) thr_sigsetmask( SIG_SETMASK, &sactions.sa_mask, NULL );
300  1    }
```

```
303
304     /*************************************************
305     **
306     **  Routine: daemon_check_proper_ID
307     **
308     **  Inputs:
309     **      None
310     **  Outputs:
311     **      None
312     **  Return Codes:
313     **      exits with an error when the user is not root
314     **  Purpose:
315     **      Checks user's ID and determines if the user is allowed
316     **      to execute service.
317     **      If there are no constraints then this
318     **      function may be blank.
319     **  Intended caller: internal only.
320     **
321     *************************************************/
322
323   void daemon_check_proper_ID()
324   {
325       /*
326  1   ** Check for root
327  1   */
328
329  1   if (geteuid() != E_ROOTUID)
330  2   {
331  2       (void) EDMRestoreEng_logent(
332  2           _FILE_, _LINE_, LOG_ERR, DAEMON_NOTSUPERUSER, 0,
333  2           "Must be run as superuser, uid was %d",
334  2           geteuid());
336  1       exit(1);
       }
```

```
338       */
339
340       /**************************************************************************
341       **
342       ** Routine: parse_commandline
343       **
344       ** Inputs:          argc, argv (command line arguments)
345       **
346       ** Outputs:         None
347       **
348       ** Return Codes:    exits with an error when the user types a bad argument
349       **
350       ** Purpose:         Parse command line arguments and sets flags. If there
351       **                  are no flags to be set then this function may be empty.
352       **
353       ** Intended caller: internal only.
354       **
355       */
357       void parse_commandline(int argc, char *argv[])
358       {
359   1       int opt;
361   1       commandlineargs = argv;
363   1       while ((opt = getopt(argc, argv, "dD")) != EOF )   /* Process options  */
364   2       {
365   2           switch(opt)
366   2           {
367   3               case 'd':
368   3               case 'D':
369   3                   G_debug = TRUE;
370   3                   debugmode = 1;          /* turn on other debugmode flag */
371   3                   break;
372   3               default:
373   3                   (void) display_usage( argv[0] );
374   3                   exit(1);
376   2           }
378   1       }
        }
```

```
380       /**************************************************************************
381       **
382       ** Routine: daemon_initialize_logging
383       **
384       ** Inputs:          None
385       **
386       ** Outputs:         None
387       **
388       ** Return Codes:    None
389       **
390       ** Purpose:         Do whatever it takes to initialize logging. In the near
391       **                  future this may involve doing something with catalogs
392       **                  or
393       **                  calling higher level logging functions which
394       **                  encapsulate
395       **                  these things.
396       **
397       ** Intended caller: internal only.
398       **
399       */
401       void daemon_initialize_logging()
402       {
403   1       /* Pass in argv[0], the program name */
404   1       (void) esl_log_init(commandlineargs[0]);
406       }
```

```
/*************************************************************
**
** Routine:  daemon_become_daemon
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**            exits with an error code if initialization fails
**
** Purpose:  This function is for doing the forking etc. under UNIX.
**           It is unknown what will be necessary under NT.
**
** Intended caller: internal only.
**
*************************************************************/

void
daemon_become_daemon()
{
    char *ptr;
    int   ret = 0;

    /*
    * Strip the path from the program name so we can use it
    * elsewhere
    */
    ptr = strrchr(commandlineargs[0], '/');
    if (ptr == NULL)
        ptr = commandlineargs[0];
    else
        ptr++;

    /* Change directory to a process specific core directory */
    ret = esl_coredir_setup(ptr);
    if (ret != 0)
    {
        (void) EDMRestoreEng_logent(__FILE__, __LINE__, LOG_ERR,
            MESSAGE_ERR_IN_ESL_COREDIR, errno,
            "esl_coredir_setup failed");

        exit(1);
    }

    /*
    ** This is now esl functionality.
    **        This code does everything necessary
    ** to make this a 'real' daemon by detaching from the
    ** changing the process group, closing stdout, stderr, stdin.
    ...
    */

    if (G_debug == FALSE)
    {
        ret = esl_daemon_startup();
        if (ret != 0)
        {
            fprintf(
                stderr, "%s: Failed to initialize as daemon.\n",
                commandlineargs[0]);
            exit(1);
        }
    }
```

```
    }
    }
}
**/
```

```
471   /*************************************************************
472    **
473    ** Routine: rpc_init
474    **
475    ** Inputs:
476    **          None
477    **
478    ** Outputs:
479    **          None
480    **
481    ** Return Codes:
482    **          exits with an error code if initialization fails
483    **
484    ** Purpose:
485    **          This function is for doing RPC initialization.
486    **          For the most part it involves calling the csc routines.
487    **          This is pretty standard between UNIX and NT.
488    **
489    ** Intended caller: internal only.
490    **
491    */
492   void rpc_init()
493   {
494      error_status_t      status;
495
496      unsigned char       *com_b;
497      struct hostent      *hp;
498      char                *ebuff;
499      struct utsname      name;
500
501      /* This is here because of HP which may or may not define timeval.
502      ** May be removed when esl_timeval is ported to clients.
503      */
504   #ifdef __STRUCT_TIMEVAL
505      struct timeval      sleep_interval = {5,0};  /* 5 second sleep interval */
506   #else
507      struct timespec     sleep_interval = {5,0};  /* 5 second sleep interval */
508   #endif
509
510      /* Setup the interface specification for RPC */
511      RR_SERVER_IFSPEC(if_spec);
512
513      /*
514      ** Login as SERVER_PRINCIPAL.  The context of the process
515      ** will be set to this principal.
516      ** This process will keep trying to login to DCE if the
517      ** server is unavailable.  Note that under SUN RPC this is a no-op.
518      */
519      while (TRUE)
520      {
521          (void) csc_server_login(RR_SERVER_PRINCIPAL,
522                                  RR_SERVER_KEYTAB, &status);
523
524          /* If we succeeded, then exit this loop. */
525          if ( status == error_status_ok )
526          {
527              break;
528          }
```

```
529          else    /* Print error message if appropriate. */
530          {
531              ebuff = (char *) csc_get_error( status );
532
534              (void) EDMRestoreEng_logent(
                        __FILE__, __LINE__, LOG_ERR,
                        MESSAGE_NO_LOGIN, 0,
                        "CSC_SERVER_LOGIN failed: <%s>",
                        ebuff ? ebuff : "Unknown error"));
537
538              /* If the failure was due to unavailable client
539               * uses sleep when SUNRPC, otherwise uses
540               * pthread call to delay for the specified
541               * time.
542               */
544              CSC_SLEEP(sleep_interval);
545              continue;
546          }
547
548          /* If we got here, we had a unexpected failure. */
550          (void) EDMRestoreEng_logent(
552              __FILE__, __LINE__, LOG_ERR,
554              MESSAGE_NO_LOGIN, 0,
555              "The service cannot log in as
556               required");
557
559          exit(1);
560      }
562
564      uname(&name);
568      hp = gethostbyname(name.nodename);
569
570      if (hp == NULL)
572      {
574          (void) EDMRestoreEng_logent(
576              __FILE__, __LINE__, LOG_ERR,
577              MESSAGE_GETHOSTNAME_FAIL, errno,
578              "gethostbyname failed" );
580
582          exit(1);
583      }
584
586      /*
587      ** We need to initialize the authorization module before we
         ** do
         ** a listener.
         */
         (void)csc_authorization_init(&status);

         if ( status != error_status_ok )
         {
             ebuff = (char *) csc_get_error( status );
             memcpy((char *) &if_spec.ip_addr, hp -> h_addr, hp->h_length);
             (void) EDMRestoreEng_logent(
                 __FILE__, __LINE__, LOG_ERR,
                 MESSAGE_NOAUTHORIZATION, 0,
```

```
588 2              "CSC_AUTHORIZATION_INIT failed: <%d> %s",
589 2              status,
590 2              ebuff ? ebuff : "Unknown error") );
591 1
591 1          exit(1);
593 1      }
595 1
595 1      conn_h = calloc(1, CONNECT_HANDLE_SIZE);
596 1
597 2      if (conn_h == NULL)
598 2      {
599 2          (void) EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
600 2              MESSAGE_NO_MEMORY, 0,
601 1              "Failure allocating memory for connection
                   handle");
602 2
604 1          exit(1);
606 1      }
607 1
609 1      (void) csc_register_private_server_interface(
610 1              0,
611 1              1,
613 2              conn_h,
614 2              &il_spec,
615 2              &status);
616 2
617 1      if ( status != error_status_ok )
618 1      {
             char *ebuff = csc_get_error( status )

             (void) EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
621 1            MESSAGE_CANNOTREGISTER, 0,
                 "CSC_REGISTER_SERVER_INTERFACE failed:
                  <%d> %s",
                 status,
                 ebuff ? ebuff : "Unknown error") );
         }

         free(conn_h);
```

```
623   ************************************************************
624   **
625   **  Routine: rpc_run
626   **
627   **  Inputs:      None
628   **
629   **  Outputs:     None
630   **
631   **  Return Codes:
632   **               None
633   **
634   **  Purpose:     This function is for running the RPC listener.
635   **               This is pretty standard between UNIX and NT.
636   **
637   **  Intended caller:  internal only.
638   **
639   ************************************************************
640   */
641   void rpc_run()
643   {
644       error_status_t    status;          /* error status (nbase.h) */
645       char  *ebuff;
646
647       /* listen for RPC calls forever. */
648       (void) csc_server_listen(
649           rpc_c_listen_max_calls_default, &status );
650       ebuff = (char *) csc_get_error( status );
652       /* We don't expect to get here. */
653       (void) EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
654           MESSAGE_SERVERLISTEN, 0,
655           "CSC_SERVER_LISTEN failed: <%d> %s",
656           status,
657           ebuff ? ebuff : "Unknown error") );

      }
```

```
659         /*
660  **
661  **     Routine:  daemon_specific_initialization
662  **
663  **     Inputs:       None
664  **
665  **     Outputs:      None
666  **
667  **     Return Codes:  None
668  **
669  **
670  **     Purpose:  Do whatever makes this daemon special.
671  **                     In some cases you
672  **               may want to start a thread or open a socket.
673  **                     Do that here.
674  **
675  **     Intended caller:  internal only.
676  **
           */
678  void
679  daemon_specific_initialization()
680  {
681  1    int           status;       /* error status (nbase.h) */
682  1    void          *statptr;
683  1    int           ret;
684  1    pthread_t     pmtid;
685  1    pthread_t     progressid;
686  1    pthread_t     ccrid;
687  1    pthread_t     ccwid;
688  1    time_t        current_time;
689  1    char          *ctimebuf;

691  1    ret = CommandAPIInit(&status);
693  1    ret = QuestionAPIInit(&status);
694  1    ret = DrainPIInit(&status);

695  1    /* Find out what time it is */
696  1    (void) time(&current_time);

698  1    ctimebuf = ctime(&current_time);

700  1    /* Overlay newline with null - but should always be 26 bytes
               long */
701  1    ctimebuf[ strlen(ctimebuf) - 1 ] = 0;

703  1    /* Log startup message */
705  1    (void) EDMRestoreEng_logmit( __FILE__, __LINE__, LOG_INFO,
                    MESSAGE_STARTUP, 0,
                    "Restore service %s starting up at %s",
                    commandlineargs[0], ctimebuf );

706  1    /*
708  1    **     Start the other threads in the daemon. The main thread
709  1    **     becomes the RPC thread. REProcessManager is the
710  1    **     entry point for the periodic event thread.
711  1    */
712  1    pthread_create(&pmtid, NULL, REProcessManager, NULL);
713  1    pthread_create(&progressid, NULL, REProgress, NULL);
714  1    pthread_create(&ccrid, NULL, RestoreSvc_ccr, NULL);
715  1    pthread_create(&ccwid, NULL, RestoreSvc_ccw, NULL);
```

```
716  1    pthread_create(&ccwtid, NULL, RestoreSvc_ccw, NULL);

719  1    rpc_init();
720  1    RestoreSvc_Setup();
721  1    rpc_run();

721  1    pthread_join(pmtid, &statptr);
724  1
     }
```

```
726  /******************************************************************
727
728  **
729  **    Routine:  daemon_cleanup
730  **
731  **    Inputs:        None
732  **
733  **    Outputs:       None
734  **
735  **    Return Codes:
736  **                   None
737  **
738  **    Purpose:       Call function which will clean up daemon properly.
739  **
740  **    Intended caller:  internal only.
741  **
742  */
743  void
744  daemon_cleanup()
745  {
746  1       {
747  1               kill_handler( 0 );
748          }
         }
```

```
 1  /*
 2  **
 3  ** Copyright 1996,1997 EMC Corporation
 4  **
 5  ** EDMProcessManager.c
 6  **
 7  ** Mission Statement: This is the entry point for the Process Manager
 8  **                    thread.
 9  **
10  ** Primary Data Acted On:
11  **
12  ** Compile-Time Options:
13  **
14  **      USE_SUNRPC - Compile source with sunrpc
15  **                   support.  If
16  **                   not set, assume DCE support.
17  **
18  ** Basic idea here: Module for coding the Process Manager thread.
19  */
20  /*
21  ** The following provides an RCS id in the binary that can be located
22  ** with the what(1) utility.  The intent is to keep this short.
23  */
24  #if defined(lint)
25  static char   RCS_id [] = "@(#)$RCSfile: EDMProcessManager.c,v $ "
26          "$Revision: 1.23 $ "
27          "$Date: 1997/02/06 20:49:15 $ ";
28  #endif
29
30  /* #define _POSIX_SOURCE       unable to compile with this define set */
31  /* #define _XOPEN_SOURCE       unable to compile with this define set */
32
33  #include <pthread.h>
34  #include <unistd.h>
35  #include <stdlib.h>
36
37  #include <esl/r_portable.h>
38  #include <esl/rp_xopen.h>
39  #include <esl/inout.h>
40
41  #include <syslog.h>
42
43  #include <EDMProcessManager.h>
44  #include <EDMRestoreApi.h>
45  #include <EDMBCommandApi.h>
46  #include <EDMRestoreEngLog.h>
47  #include <EDMmain.h>
48  #include <restore/restore_engine.h>
49  #include <restore/restore_api.h>
50  #include <restore/REprocmsg.h>
51  #include <restore/EDMREprogressApi.h>
52  #include <EDMPlimStatus.h>
53
54  /* local prototypes */
55  static void unregister_rpc( void );
56  static void start_completion( EDMREGlobalStatus );
57
58  /* local data */
59  static boolean_ty    completion_signalled = FALSE;
60
61  struct timeout_array
62  {
63      time_t           timeout_len;
64      time_t           guidesdlen;
```

```
 65  } tval[ MAX_GLOBAL_STATUS_VALUE+1 ] =
 66  {
 67    1,                        1,     // Idle  Gui is dead
 68    5*SECONDS_PER_MINUTE,     5*SECONDS_PER_MINUTE,   // Timeout value
 69    5*SECONDS_PER_MINUTE,     5*SECONDS_PER_MINUTE,   // Exiting
 70    5*SECONDS_PER_MINUTE,     5*SECONDS_PER_MINUTE,   // Starting
 71    3*SECONDS_PER_HOUR,       3*SECONDS_PER_HOUR,     // Browsing
 72    2*SECONDS_PER_YEAR,       2*SECONDS_PER_YEAR,     // Pre phase
 73    10*SECONDS_PER_DAY,       10*SECONDS_PER_DAY,     // Executing
 74    2*SECONDS_PER_DAY,        2*SECONDS_PER_DAY,      // Post phase
 75    5*SECONDS_PER_DAY,        5*SECONDS_PER_DAY,
 76  };
 77
 78
 79  boolean_ty
 80  IsRestoreTimedOut( IN time_t lasttime,  IN time_t incurrentstate,
 81                     IN int status)
 82  {
 83      time_t t = time(NULL);
 84
 85      if ( status < 0 )
 86          return FALSE;
 87
 88      if (status > MAX_GLOBAL_STATUS_VALUE)
 89          return FALSE;
 90
 91      status = 0;         /* all exiting conditions use same timeout */
 92
 93      if ( ( t - tval[status].guidesdlen) > lasttime)
 94          return TRUE;
 95      else if ( ( t - tval[status].timeout_len) > incurrentstate)
 96          return TRUE;
 97
 98      return FALSE;
 99  }
100
101
102
103
```

```
105  void *
106  REProcessManager(void *buff)
107  {
108      int         status;
109      int         command;
110      void        *input_ptr;
111      void        *output_ptr;
112      boolean_ty  result;
113      boolean_ty  finish_rpc_recvd = FALSE;
114      boolean_ty  reader_finish_recvd = FALSE;
115      EDMRESGlobalStatus  internal_status;
116      EDMRESGlobalStatus  status_time;
117
118      setGlobalStatus( EDMRE_STATE_STARTING );
119
120      while
121      (reader_finish_recvd || !finish_rpc_recvd )      /* until time to exit */
122      {
123          /* wait for next command */
124          if (PopCommand( 1, &command, &status )
125          {
126              /* queue empty or got error */
127              if (COMMAND_RECORD_GET_FAILED != status)
128              {                          /* report error */
129                  EDMRestoreEng.logent(  _FILE_,  _LINE_, LOG_ERR,
130                      MESSAGE_RECORD_GET_FAILED, 0,
131                      "PopCommand failed",
132                      status );
133              }
134
135              /* let restore service module clean up, stop rpcs */
136              internal_status = getGlobalStatus( &status_time );
137              if (completion_signalled)
138              {
139                  if (TRUE == IsRestoreTimedOut(
140                          internal_status )
141                  {
142                      if (!finish_rpc_recvd)
143                      {
144                          unregister_rpc( );    /* cleanup csc i/f */
145                          EDMRestoreEng.logent(  _FILE_, _LINE_, LOG_ERR,
146                              MESSAGE_SHUTDOWN, 0,
147                              "Shutting down after timeout waiting",
148                              internal_status );
149                      }
150                      break;      /* escape while to exit */
151                  }
152              }
153              else
154                  start_completion( internal_status );
155
156              continue;
157              /* keep waiting in case thread wait got interrupted */
158          }
159
160      }
```

```
161      /* got some command; see if we're in completion sequence: */
162      if (completion_signalled)
163      {
164          if (COMMAND_FINISH == command && !reader_finish_recvd )
165          {
166              if (PopRpcInput( &input_ptr, &status ))
167              {
168                  EDMRestoreEng.logent(  _FILE_, _LINE_, LOG_ERR,
169                      MESSAGE_POP_RPC_INPUT_FAILED, 0,
170                      "PopRpcInput failed",
171                      status = %d", status );
172              }
173              else
174              {
175                  /* let restore service module clean up; */
176                  result = EDMRE_Finish( input_ptr, &output_ptr );
177
178                  finish_rpc_recvd = TRUE;
179                  unregister_rpc( );      /* cleanup csc i/f */
180              }
181          }
182          else if (
183              COMMAND_READER_FINISHED == command && !reader_finish_recvd)
184          {
185              reader_finish_recvd = TRUE;
186              if (finish_rpc_recvd)
187              {
188                  /* let restore service module clean up, stop rpcs */
189                  result = EDMRE_Finish( NULL, NULL );
190                  unregister_rpc( );      /* cleanup csc i/f */
191                  break;      /* exit */
192              }
193          }
194          else
195          {
196              EDMRestoreEng.logent(  _FILE_, _LINE_, LOG_ERR,
197                  MESSAGE_INVALID_COMMAND, 0,
198                  "cmd value: %d",
199                  command );
200          }
201
202          continue;
203          /* check if both finishes recvd, else keep waiting */
204      }
205
206      /* not in completion seq;
207         get pointer to rpc input argument structure */
208      if (PopRpcInput( &input_ptr, &status ))
209      {
210          EDMRestoreEng.logent(  _FILE_, _LINE_, LOG_ERR,
211              MESSAGE_POP_RPC_INPUT_FAILED, 0,
212              "PopRpcInput failed; status = %d Lost command: %d",
213              status, command );
214
215          continue;   /* ??? keep trying or return ?? */
216      }
217
218      switch ( command )     /* %d */
219      {
220          case COMMAND_GET_RESTORABLE_OBJECTS:
221              result = EDMRE_GetRestorableObjects(
222                  input_ptr, &output_ptr );
223              break;
224          case COMMAND_MARK_OBJECT:
225              result = EDMRE_MarkObject(  input_ptr, &output_ptr );
226              break;
```

```
218  3          case COMMAND_UNMARK_OBJECT:
219  3              result = EDMRE_UnmarkObject( input_ptr );
220  3              break;
221  3          case COMMAND_SUBMIT:
222  3              result = EDMRE_Submit( input_ptr, &output_ptr );
223  3              break;

225  3          case COMMAND_START:
226  3              result = EDMRE_Start( input_ptr, &output_ptr );
227  3              break;
          /* taken out to allow configuration after successful & aborted
                                                       restore */
228  #if 0
228  4              start_completion( getGlobalStatus (NULL) );  /* leave same state */
230  3              break;
232  3  #endif

234  3          case COMMAND_FIND_RESTORABLE_OBJECTS:
                    result = EDMRE_FindRestorableObjects(
                                 input_ptr, &output_ptr );
236  3              break;

237  3          case COMMAND_FINISH:
                    result = EDMRE_Finish( input_ptr, &output_ptr );
                    finish_rpc_recvd = TRUE;
                    if ( EDMRE_STATE_SUCCESSFUL
                         < (internal_status = getGlobalStatus(
                                                   &status_time )) )
241  3              /* if already exiting, leave state alone */
                        start_completion( EDMRE_STATE_SUCCESSFUL );
                    else
                        start_completion( internal_status );
                    unregister_rpc();
247  3              break;

249  3          case COMMAND_LOAD_RECX_DIRECTIVES:
                    result = EDMRE_Load_recx_directives(
                                 input_ptr, &output_ptr );
251  3              break;
              case COMMAND_GET_ALL_TIMES:
                    result = EDMRE_GetAllBackupTimes(
                                 input_ptr, &output_ptr );
253  3              break;
254  3          case COMMAND_SET_NEXT_BACKUP:
                    result = EDMRE_SetNextBackup( input_ptr, &output_ptr );
                    break;
256  3          case COMMAND_SET_PREVIOUS_BACKUP:
                    result = EDMRE_SetPreviousBackup(
                                 input_ptr, &output_ptr );
                    break;
259  3          case COMMAND_SET_FIRST_BACKUP:
                    result = EDMRE_SetFirstBackup( input_ptr, &output_ptr );
                    break;
262  3          case COMMAND_SET_MOST_RECENT_BACKUP:
                    result = EDMRE_SetMostRecentBackup(
                                 input_ptr, &output_ptr );
                    break;
265  3          default:
                    EDMRestoreEng_logent( __FILE__, __LINE__, __LOG_ERR,
                                MESSAGE_INVALID_COMMAND, 0,
                                "cmd value: %d",
                                command );
                    result = COMMAND_RESULT_FAILURE;
```

```
277  2          /* push result arg structure pointer, IF command succeeded */
278  2          if ( result != COMMAND_RESULT_FAILURE)
279  3              if (PushRpcOutput( output_ptr, &status))
280  3                  EDMRestoreEng_logent( __FILE__, __LINE__, __LOG_ERR,
                                MESSAGE_PUSH_RPC_OUTPUT_FAILED,
                                0,
                                "PushRpcOutput failed:
                                status = %d",
                                status );

287  3          if (PushResult( result, command, &status) )
288  3              EDMRestoreEng_logent( __FILE__, __LINE__, __LOG_ERR,
290  2                          MESSAGE_PUSH_RESULT_TO_QUEUE_RESULT, 0,
                                "PushResult failed:
                                status = %d", status );
292  3          }

293  3  #if 0
294  3          /* I think we just leave global status as its already set*/
295  1          if ( reader_finish_rcvd && finish_rpc_recvd )
                                                    /* good exit ?? */
298  1              setGlobalStatus( /* good exit */
299  1                               getGlobalStatus (NULL) );
300  1  #endif

302  1          exit ( getGlobalStatus(NULL) );
303  1          return buf;
304  1      }
```

```
307    /* local function to unregister rpc interface */
308
309    static void unregister_rpc( void )
310  1 {
311  1     sleep( 1 );       /* allow last rpc (finish) response to get sent */
312  1     unregister_csc( );   /* stop RPC traffic */
313  1     return;
314    }
```

```
317    /* local function to start completion sequence */
318
319    static void start_completion( EDMREGlobalStatus status )
320  1 {
321  1     setGlobalStatus( status );
322  1     sendFinalStatus( );
323  1     completion_signaled = TRUE;    /* signal dispatcher */
324  1     return;
325    }
```

```
  2   /****************************************************************
  3   **
  4   **
  5   **
  6   **   Copyright (c) 1998,1999 by EMC Corporation.
  7   **
  8   **   File Name:  EDMREProcMgrService.c
  9   **
 10   **   Purpose:  This module contains the top level processing of the
 11   **             Process Manager (thread) functions that
 12   **             provide the top level processing of the "asynchronous" Restore
 13   **             Restore Service library calls that perform the actual RPC
 14   **             Engine RPC's.  These functions are basically 'wrappers' for the
 15   **             services.
 16   **
 17   **   Table of Contents:
 18   **
 19   **          EDMRE_GetRestorableObjects
 20   **          EDMRE_MarkObject
 21   **          EDMRE_UnmarkObject
 22   **          EDMRE_Submit
 23   **          EDMRE_Start
 24   **          EDMRE_Finish
 25   **          EDMRE_FindRestorableObjects
 26   **
 27   **       Internal Functions:
 28   **          EDMRE_ProgressCallback
 29   **          EDMRE_RestoreCallback
 30   **
 31   **   Compile-Time Options:
 32   **
 34   **   The following provides an RCS id in the binary that can be located
 35   **   with the what(1) utility.  The intent is to keep this short.
 36   **
 38   #ifndef lint
 39   static char RCS_id [] =
 40       "$RCSfile$ "
 41       "$Revision$ "
 42       "$Date$";
 42   #endif
 45   /* Feature test switches.
 46   **
 47   **   Standard defines required to turn on OS features go here.
 48   **
 49   **   The following is required for code that uses POSIX API's.
 50   **   Standard defines required to turn on OS features go here.
 51   **   Remove for non-POSIX, non-portable code.
 53   /* #define _POSIX_SOURCE 1 */
 55   /* System headers.
 56   **
 57   **   System headers.
 59   #include <stdlib.h>
 60   #include <sys/syslog.h>
```

```
 62   #include <unistd.h>
 63   #include <string.h>
 65   /*
 66   **   Epoch headers.
 67   */
 68   #include <ebutil/ebutil.h>
 69   #include <restore/restore_engine.h>
 70   #include <RSapi.h>
 71   #include <EDMREProcessManager.h>
 72   #include <restore/RBprogmsg.h>
 73   #include <EDMRestoreEng.h>
 74   #include <EDMRECommandApi.h>
 76   /*
 77   **   Local headers.
 78   */
 82   static boolean_t EDMRE_ProgressCallback( unsigned long long progress );
 84   static boolean_t EDMRE_RestoreCallback( void );
 88   /*
 89   **
 90   **   Routine:   EDMRE_GetAllBackupTimes
 91   **
 92   **   Purpose:  Wrapper of Restore service library call to be executed
 93   **             asynchronously from main RPC thread
 94   **
 95   **   Inputs:   void *input_args    ptr to struct with RPC input args
 96   **                                 arg struct
 97   **   Outputs:  void **status       addr of void * to receive ptr output
 98   **                                 arg struct
 99   **
100   **   Return Codes:
101   **       0 for success and non-zero for failure.
102   **
103   */
104   int EDMRE_GetAllBackupTimes( void *input_args, void **output_args )
105   {
107       RE_get_all_backup_times_args      *in_args
108       RE_get_all_backup_times_result    *out_args;
110       int     status = COMMAND_RESULT_SUCCESS;
112       in_args  = (RE_get_all_backup_times_args *)input_args;
115       out_args = calloc( 1, sizeof (RE_get_all_backup_times_result) );
116       if (NULL == out_args)
117       {
118           EDMRestoreEng_logent(
118               __FILE__, __LINE__, LOG_ERR, MESSAGE_NO_MEMORY,
120               0, "calloc fail for RE_get_all_backup_times_args" );
121           status = COMMAND_RESULT_FAILURE;    /* fatal error */
122       }
122       else
```

```
123 2   {
124 2       out_args->cookie = in_args->cookie;
125 2       out_args->status = RSTSI_GetAllBackupTimes(
126 2                              in_args->startTime,
127 2                              in_args->endTime,
128 2                              in_args->maxEntries,
129 2                              in_args->flags,
130 2                              &out_args->backupTimes,
131 2                              &out_args->numEntries,
132 2                              &out_args->cookie);
133 2
134 2       *output_args = (void *) out_args;
135 1
136 1       xdr_free( xdr_RE_get_all_backup_times_args, (char *)in_args);
137 1       free( in_args);
138 1
139         return status;
    }
```

```
141     */
142     /*********************************************************
143      **
144      ** Routine:    EDMRE_GetRestorableObjects
145      **
146      ** Inputs:     void *input_args       ptr to struct with RPC input args
147      **                                    &out_args->childObjs
148      ** Outputs:    void *status           addr of void * to receive ptr output
149      **                                    arg struct
150      **
151      ** Return Codes:
152      **              0 for success and non-zero for failure.
153      **
154      ** Purpose:    Wrapper of Restore service library call to be executed
155      **             asynchronously from main RPC thread
156      **
157      *********************************************************/
158 1   int
159 1   EDMRE_GetRestorableObjects( void *input_args, void **output_args )
160 1   {
161 1       RE_get_restorable_objects_start_args   *in_args
162 1               = (RE_get_restorable_objects_start_args *)input_args;
163 1       RE_get_restorable_objects_output_result *out_args;
164 1
165 1       int        status = COMMAND_RESULT_SUCCESS;
166 1
167 2       out_args = calloc( 1, sizeof(
168 2                  RE_get_restorable_objects_output_result) );
169 2
170 2       if (NULL == out_args) {
171 1           EDMRestoreEng_logent(
172 2               __FILE__, __LINE__, LOG_ERR, MESSAGE_NO_MEMORY,
173 1               0,
174 1               "calloc fail for RE_get_restorable_objects_output_result" );
175 2           status = COMMAND_RESULT_FAILURE;     /* fatal error */
176 2       }
177 1       else
178 2       {
179 2           out_args->cookie = in_args->cookie;
180 2           out_args->status = RSTSI_GetRestorableObjects(
181 2               (restorableObjectPtr)in_args->parentObj )->RE_restorable_obj_u.
182 2                        ticInfo,
183 2               in_args->parentObj->objLevel,
184 2               &out_args->childObjs,
185 2               out_args->cookie,
186 2               in_args->maxEntries,
187 1               &out_args->numEntries,
188 1               in_args->allowBadFiles );
189 1
190 1           *output_args = (void *)out_args;
191 1       }

        xdr_free( xdr_RE_get_restorable_objects_start_args, (
                       char *)in_args);
        free( in_args);

        return status;
    }
```

```
/*
**
** Routine:    EDMRE_MarkObject
**
** Inputs:     void *input_args    ptr to struct with RPC input args
**
**                                 addr of void * to receive ptr output
**                                                              arg struct
** Outputs:    void **output_args
**
** Return Codes:
**                0 for success and non-zero for failure.
**
** Purpose:    Wrapper of Restore service library call to be executed
**             asynchronously from main RPC thread
**
*/
int
EDMRE_MarkObject( void *input_args, void *output_args )
{
    RE_get_mark_results_result  *out_args;
    in_args = (
        RE_mark_object_args *)input_args;

    status = COMMAND_RESULT_SUCCESS;

    out_args = calloc( 1, sizeof(RE_get_mark_results_result) );
    if (NULL == out_args)
    {
        EDMReStoreEng_logent(
            __FILE__,       __LINE__,   LOC_ERR,  MESSAGE_NO_MEMORY,
            0, "calloc fail for RE_get_mark_results_result" );
        status = COMMAND_RESULT_FAILURE;    /* fatal error */
    }
    else
    {
        out_args->status = RSTSL_MarkObject( in_args->thisObj,
            in_args->backupTime,
            in_args->allowBadFiles,
            in_args->descend,
            in_args->badFileCount,
            &out_args->permDenyFileCount,
            &out_args->fileMarkCount,
            &out_args->dirMarkCount,
            &out_args->otherMarkCount,
            EDMRE_ProgressCallback );

        *output_args = (void *)out_args;
    }

    xdr_free( xdr_RE_mark_object_args, (char *)in_args);
    free( in_args );

    return status;
}
```

```
/*
**
** Routine:    EDMRE_UnmarkObject
**
** Inputs:     void *input_args    ptr to struct with RPC input args
**
**                                 addr of void * to receive ptr output
**                                                              arg struct
** Outputs:    void **output_args
**
** Return Codes:
**                0 for success and non-zero for failure.
**
** Purpose:    Wrapper of Restore service library call to be executed
**             asynchronously from main RPC thread
**
*/
int
EDMRE_UnmarkObject( void *input_args, void *output_args )
{
    RE_get_unmark_results_result  *out_args;
    in_args = (
        RE_unmark_object_args *)input_args;

    status = COMMAND_RESULT_SUCCESS;

    out_args = calloc( 1, sizeof(RE_get_unmark_results_result) );
    if (NULL == out_args)
    {
        EDMReStoreEng_logent(
            __FILE__,       __LINE__,   LOC_ERR,  MESSAGE_NO_MEMORY,
            0, "calloc fail for RE_get_unmark_results_result" );
        status = COMMAND_RESULT_FAILURE;    /* fatal error */
    }
    else
    {
        out_args->status = RSTSL_UnmarkObject( in_args->thisObj,
            in_args->backupTime,
            in_args->allowBadFiles,
            in_args->descend,
            in_args->badFileCount,
            &out_args->badFileCount,
            &out_args->fileMarkCount,
            &out_args->dirMarkCount,
            &out_args->otherMarkCount,
            EDMRE_ProgressCallback );

        *output_args = (void *)out_args;
    }

    xdr_free( xdr_RE_unmark_object_args, (char *)in_args);
    free( in_args );

    return status;
}
```

```c
296  /**********************************************************
297  **
298  ** Routine:   EDMRE_ProgressCallback
299  **
300  ** Inputs:    unsigned long progress      objects processed so far
301  **
302  ** Outputs:   none
303  **
304  ** Return Codes:
305  **    boolean_ty
306  **                       FALSE if operation can continue
307  **                       TRUE if operation should be cancelled
308  **
309  ** Purpose:   Restore service library callback function to be called
310  **            to return progress information and check for
311  **                                                    cancellation.
312  **
313  **********************************************************/
312  */
313  static boolean_ty EDMRE_ProgressCallback( unsigned long progress )
314 1 {
315 1    UpdateProgressValue( progress );
316 1    return TestRpcCancelFlag( );
317 1 }
```

```c
319  /**********************************************************
320  **
321  ** Routine:   EDMRE_RestoreCallback
322  **
323  ** Inputs:    none
324  **
325  ** Outputs:   none
326  **
327  ** Return Codes:
328  **    boolean_ty
329  **                       FALSE if operation can continue
330  **                       TRUE if operation should be cancelled
331  **
332  ** Purpose:   Restore service library callback function to be called
333  **            by 'Start' function to return check for cancellation.
334  **
335  **********************************************************/
335  */
336  static boolean_ty EDMRE_RestoreCallback( void )
337 1 {
338 1    long               last_rpc_time;
339 1    time_t             status_time;
340 1    EDMREGlobalStatus  internal_status;
341 1
342 1    internal_status = getGlobalStatus( &status_time );
343 1    last_rpc_time = getLastRpcTime( );
344 1
345 1    if ( internal_status == EDMRE_STATE_USER_QUIT      /* someone */
346 1       || internal_status == EDMRE_STATE_ADMIN_QUIT)   /* aborted */
347 1       return TRUE;
348 1    if (TRUE == IsRestoreTimedOut( last_rpc_time, status_time,
349 1                                    internal_status )
350 1       return TRUE;                          /* no sign of user life */
351 1    }
352 1    return TestRpcCancelFlag( );          /* user-signaled cancel */
353  }
```

```
355
356  **
357  ** Routine:     EDMRE_Submit
358  **
359  ** Inputs:      void *input_args        ptr to struct with RPC input args
360  **
361  ** Outputs:     void *status            addr of void * to receive ptr for output
362  **                                      arg struct
363  **
364  ** Return Codes:
365  **              0 for success and non-zero for failure.
366  **
367  ** Purpose:     Wrapper of Restore service library call to be executed
368  **              asynchronously from main RPC thread
369  **
370  */
371  int EDMRE_Submit( void *input_args, void *output_args )
372  {
373      RE_get_submit_args      *in_args = (
374      RE_submit_results_output    *)input_args;
375      unsigned int            object_count = 0;
376      EDMRE_submit_args       *submitArgs = calloc(1, sizeof(
377      int             status = COMMAND_RESULT_SUCCESS;

379      out_args = calloc( 1, sizeof(RE_get_submit_results_output) );
380      if (NULL == out_args)
382      {
383          EDMRestoreEng_logout(
384              __FILE__, __LINE__, LOG_ERR, MESSAGE_NO_MEMORY,
385              0, "calloc fail for RE_get_submit_results_output" );
386          status = COMMAND_RESULT_FAILURE;      /* fatal error */
387      }
388      else
389      {
390          submitArgs->socketClientName = in_args->socketClientName;
391          submitArgs->clientSocketPort = in_args->socketPort;
393          submitArgs->hostname = in_args->hostname;
395          submitArgs->mapfile_env = el_strdup(in_args->mapfile_env,
396              in_args->overwritePolicy,
397              in_args->inPlace,
398              in_args->transport,
399              in_args->directory,
400              in_args->submitObjectID,
401              &object_count,
402              &object_count,
403              submitArgs);
405          out_args->objectsDone = object_count;
406          out_args->status

              EDMRE_ProgressCallback,
      }

407      xdr_free( xdr_RE_submit_args, (char *)in_args );
408      free( in_args );

410      return status;
411  }
```

```
413
414  **
415  ** Routine:     EDMRE_Start
416  **
417  ** Inputs:      void *input_args        ptr to struct with RPC input args
418  **
419  ** Outputs:     void *status            addr of void * to receive ptr for output
420  **                                      arg struct
421  **
422  ** Return Codes:
423  **              0 for success and non-zero for failure.
424  **
425  ** Purpose:     Wrapper of Restore service library call to be executed
426  **              asynchronously from main RPC thread
427  **
428  */
429  int EDMRE_Start( void *input_args, void **output_args )
430  {
431      RE_start_args       *in_args = (RE_start_args *)input_args;
432      RE_status_result    *out_args;
434      int         status = COMMAND_RESULT_SUCCESS;

436      out_args = calloc( 1, sizeof(RE_status_result) );
437      if (NULL == out_args)
438      {
439          EDMRestoreEng_logout(
440              __FILE__, __LINE__, LOG_ERR, MESSAGE_NO_MEMORY,
441              0, "calloc fail for RE_status_result" );    /* fatal error */
442          status = COMMAND_RESULT_FAILURE;
444      }
445      else
446      {
447          out_args->status = RSTSL_Start( in_args->submitObjectID,
448              EDMRE_RestoreCallback );
449          *output_args = (void *)out_args;
452      }

454      xdr_free( xdr_RE_start_args, (char *)in_args );
455      free( in_args );

         return status;
     }
```

```c
457  /*
458  ******************************************************************
459  **
460  ** Routine:     EDMRE_Finish
461  **
462  ** Inputs:      void *input_args      ptr to struct with RPC input
463  **                                    args
464  **
465  ** Outputs:     void **status         OPTIONAL addr of void * to receive
466  **                                    ptr
467  **
468  ** Return Codes:
469  **              0 for success and non-zero for failure.
470  **
471  ** Purpose:     Wrapper of Restore service library call to be executed
472  **              asynchronously from main RPC thread
473  **
474  ******************************************************************
475  */
476
477  int EDMRE_Finish( void *input_args, void **output_args )
478  {
479      int              status = COMMAND_RESULT_SUCCESS;
480                       RE_status_result    *out_args;
481
482      status = RSTSL_Finish( );
483
484      xdr_free( xdr_RE_null_args, (char *)input_args );
485      free( input_args );
486
487      if ( (out_args = calloc( 1, sizeof(RE_status_result) )) != E_SUCCESS)
488          *output_args = (void *)out_args;
489      else
490          /* only keep output struct if user want it */
491          free( out_args );
492
493      return status;
494  }
```

```c
496  /*
497  ******************************************************************
498  **
499  ** Routine:     EDMRE_FindRestorableObjects
500  **
501  ** Inputs:      void *input_args      ptr to struct with RPC input args
502  **
503  ** Outputs:     void **status         addr of void * to receive ptr for output
504  **                                    arg struct
505  **
506  ** Return Codes:
507  **              0 for success and non-zero for failure.
508  **
509  ** Purpose:     Wrapper of Restore service library call to be executed
510  **              asynchronously from main RPC thread
511  **
512  ******************************************************************
513  */
514
515  int EDMRE_FindRestorableObjects(  void *input_args, void **output_args )
516  {
517      int              status = COMMAND_RESULT_SUCCESS;
518                       RE_find_restorable_objects_args     *in_args;
519                       RE_find_restorable_objects_result   *out_args;
520                       EDMRE_SearchCriteriaRec             searchCriteria;
521
522      if ( (out_args = calloc( 1, sizeof(
523                  RE_find_restorable_objects_result) )) )
524                  ;
525      else
526      {
527          EDMREsourcelog_logent( FILE__, LINE__, LOG_ERR,
528                  "calloc fail for
529                  RE_find_restorable_objects_result"  MESSAGE_NO_MEMORY, 0,
530          status = COMMAND_RESULT_FAILURE;    /* fatal error */
531      }
532
533      /* prepare search criteria structure for input to
534         RSTST func: */
535      searchCriteria.descendDirectory =
536          in_args->searchCriteria.descendDirectory;
537      searchCriteria.startDirectory =
538          in_args->searchCriteria.startDirectory;
539      strcpy( searchCriteria.typeOfFile =
540          in_args->searchCriteria.typeOfFile);
541      strcpy( searchCriteria.owner,
542          in_args->searchCriteria.owner, 64);
543      strcpy( searchCriteria.searchString,
544          in_args->searchCriteria.searchString, 128);
545      searchCriteria.excludeOwner =
546          in_args->searchCriteria.excludeOwner;
547      strcpy( searchCriteria.group,
548          in_args->searchCriteria.group, 64);
```

```
549         searchCriteria.excludeGroup =
550             in_args->searchCriteria->excludeGroup;
551         searchCriteria.sizeInBytes.high =
552             in_args->searchCriteria->sizeInBytes.high;
553         searchCriteria.sizeInBytes.low =
554             in_args->searchCriteria->sizeInBytes.low;
555         searchCriteria.sizeMatch =
556             in_args->searchCriteria->sizeMatch;
557         searchCriteria.startTime =
558             in_args->searchCriteria->startTime;
559         searchCriteria.endTime =
560             in_args->searchCriteria->endTime;
561
562         out_args->status = RSTSL_findRestorableObjects(
563             &searchCriteria,
564             EDMRE_ProgressCallback );
565
566         *output_args = (void *)out_args;
567
568         xdr_free( xdr_RE_find_restorable_objects_args,
569             (char *)in_args );
570         free( in_args );
571         return status;
572     }
```

```
576     /************************************************************
577      **
578      ** Routine:     int EDMRE_load_recx_directives
579      **
580      ** Inputs:      RE_recx_file_info *fileinfo   Information on file to be
581      **                                            retrieved
582      **
583      ** Outputs:     Error or success from the RSTSL call
584      **
585      ** Return Codes:
586      **              0 for success and non-zero for failure.
587      **
588      ** Purpose:     Function to retrieve directives file from client and then
589      **              load the file contents into the context structure.  The file
590      **
591      **              transfer is done with edm link.
592      **
593      **
594      *************************************************************/
595
596     int EDMRE_load_recx_directives( void *input_args,
597                                     void *output_args
598                                     )
599     {
600         RE_status_result *outargs;
601         RSTRPC_recx_file_info * fileinfo = (
602             RSTRPC_recx_file_info *)input_args;
603         outargs = calloc(1,sizeof(RE_status_result));
604
605         /*
606          * Actually load the recx structure.
607          */
608         outargs->status = RSTSL_load_recx_directives(fileinfo);
609
610         /*
611          * Return that the RPC was atleast successful,
                the load may not have been
                */
                return COMMAND_RESULT_SUCCESS;
         }
```

```
614
615   /*
616   **********************************************************
617   **
618   ** Routine:    EDMRE_SetPreviousBackup
619   **
620   ** Inputs:     void *input_args      ptr to struct with RPC input args
621   **
622   ** Outputs:    void **status         addr of void * to receive ptr output
623   **                                   arg struct
624   **
625   ** Return Codes:
626   **             0 for success and non-zero for failure.
627   **
628   ** Purpose:    Wrapper of Restore service library call to be executed
629   **             asynchronously from main RPC thread
630   **
631   **********************************************************
632   */
633   int  EDMRE_SetPreviousBackup( void *input_args, void **output_args )
634   {
636   1      RE_set_backup_time_args      *in_args
                = (RE_set_backup_time_args *)input_args;
638   1      RE_get_all_backup_time_result *out_args;
640   1      int          status = COMMAND_RESULT_SUCCESS;
641   1
642   1      out_args = calloc( 1, sizeof (RE_status_result) );
643   1
644   1      if (NULL == out_args)
645   2      {
646   2          EDMRestoreEng_logent(
                     FILE__, LINE__, LOG_ERR, MESSAGE_NO_MEMORY,
                     0, "calloc fail for RE_status_result" );
648   2          status = COMMAND_RESULT_FAILURE;    /* fatal error */
649   2      }
650   2      else
651   2      {
652   2          out_args->status = RSTSL_SetPreviousBackup( in_args->flags);
653   2          *output_args = (void *) out_args;
654   1      }
655   1
657   1      xdr_free( xdr_RE_set_backup_time_args, (char *)in_args);
658   1      free( in_args);

          return status;
      }
```

```
662   /*
663   **********************************************************
664   **
665   ** Routine:    EDMRE_SetBackupForTime
666   **
667   ** Inputs:     void *input_args      ptr to struct with RPC input args
668   **
669   ** Outputs:    void **status         addr of void * to receive ptr output
670   **                                   arg struct
671   **
672   ** Return Codes:
673   **             0 for success and non-zero for failure.
674   **
675   ** Purpose:    Wrapper of Restore service library call to be executed
676   **             asynchronously from main RPC thread
677   **
678   **********************************************************
679   */
680   int  EDMRE_SetBackupForTime( void *input_args, void **output_args )
681   {
682   1      RE_backup_for_time_args      *in_args
                = (RE_backup_for_time_args *)input_args;
684   1      RE_get_all_backup_times_result *out_args;
686   1      int          status = COMMAND_RESULT_SUCCESS;
689   1
690   1      out_args = calloc( 1, sizeof (RE_status_result) );
691   1
692   2      if (NULL == out_args)
693   2      {
694   2          EDMRestoreEng_logent(
                     FILE__, LINE__, LOG_ERR, MESSAGE_NO_MEMORY,
                     0, "calloc fail for RE_status_result" );
695   2          status = COMMAND_RESULT_FAILURE;    /* fatal error */
696   2      }
698   2      else
699   2      {
700   2          out_args->status = RSTSL_SetBackupForTime( in_args->time,
                                                            in_args->flags);
702   1          *output_args = (void *) out_args;
703   1      }
705   1
706   1      xdr_free( xdr_RE_backup_for_time_args, (char *)in_args);
          free( in_args);

          return status;
      }
```

```c
708      */
709  /*****************************************************************
710  **
711  ** Routine:    EDMRE_SetNextBackup
712  **
713  ** Inputs:     void *input_args    ptr to struct with RPC input args
714  ** Outputs:    void **status       addr of void * to receive ptr output
                                        arg struct
716  **
717  ** Return Codes:
718  **     0 for success and non-zero for failure.
719  **
720  ** Purpose: Wrapper of Restore service library call to be executed
721  **          asynchronously from main RPC thread
722  **
723  **************************************************************** */
723  int EDMRE_SetNextBackup( void *input_args, void **output_args )
724  {
725      RE_set_backup_time_args           *in_args;
726      RE_set_backup_time_args           *in_args = (RE_set_backup_time_args *)input_args;
727      RE_get_all_backup_times_result    *out_args;
728
730      int        status = COMMAND_RESULT_SUCCESS;
732      out_args = calloc( 1, sizeof (RE_status_result) );
735      if (NULL == out_args)
736      {
737          EDMRestoreEng_logent(
738              __FILE__,  __LINE__,  LOG_ERR, MESSAGE_NO_MEMORY,
739              0, "calloc fail for RE_status_result" );
740          status = COMMAND_RESULT_FAILURE;  /* fatal error */
741      }
742      else
743      {
744          out_args->status = RSTSL_SetNextBackup( in_args->flags);
745          *output_args = (void *) out_args;
746      }
748      xdr_free( xdr_RE_set_backup_time_args, (char *)in_args);
749      free( in_args);
751      return status;
752  }
```

```c
754  /*****************************************************************
755  **
756  ** Routine:    EDMRE_SetFirstBackup
757  **
758  ** Inputs:     void *input_args    ptr to struct with RPC input args
759  ** Outputs:    void **status       addr of void * to receive ptr output
                                        arg struct
761  **
762  ** Return Codes:
763  **     0 for success and non-zero for failure.
764  **
765  ** Purpose: Wrapper of Restore service library call to be executed
766  **          asynchronously from main RPC thread
767  **
768  **************************************************************** */
769  int EDMRE_SetFirstBackup( void *input_args, void **output_args )
770  {
771      RE_set_backup_time_args           *in_args;
772      RE_set_backup_time_args           *in_args = (RE_set_backup_time_args *)input_args;
773      RE_get_all_backup_times_result    *out_args;
774
776      int        status = COMMAND_RESULT_SUCCESS;
778      out_args = calloc( 1, sizeof (RE_status_result) );
781      if (NULL == out_args)
782      {
783          EDMRestoreEng_logent(
784              __FILE__,  __LINE__,  LOG_ERR, MESSAGE_NO_MEMORY,
785              0, "calloc fail for RE_status_result" );
786          status = COMMAND_RESULT_FAILURE;  /* fatal error */
787      }
788      else
789      {
790          out_args->status = RSTSL_SetFirstBackup( in_args->flags);
791          *output_args = (void *) out_args;
792      }
794      xdr_free( xdr_RE_set_backup_time_args, (char *)in_args);
795      free( in_args);
797      return status;
798  }
```

```c
800
801   /*
802    ****************************************************
803    **
804    **  Routine:    EDMRE_SetMostRecentBackup
805    **
806    **  Inputs:     void *input_args    ptr to struct with RPC input args
807    **
808    **  Outputs:    void *status        addr of void * to receive ptr output
809    **                                  arg struct
810    **
811    **  Return Codes:
812    **              0 for success and non-zero for failure.
813    **
814    **  Purpose:    Wrapper of Restore service library call to be executed
815    **              asynchronously from main RPC thread
816    **
817    ****************************************************
818    */
815   int  EDMRE_SetMostRecentBackup( void *input_args, void *output_args )
816 1 {
817 1     RE_set_backup_time_args          *in_args;
818 1     RE_set_backup_time_args          = (RE_set_backup_time_args *)input_args;
819 1     RE_get_all_backup_times_result   *out_args;
820 1
822 1     int        status = COMMAND_RESULT_SUCCESS;
824 1     out_args = calloc( 1, sizeof (RE_status_result) );
827 1     if (NULL == out_args)
828 2     {
829 2         EDMRestoreEng_logerr(
830 2             _FILE_, _LINE_, LOG_ERR, MESSAGE_NO_MEMORY,
831 2             0, "calloc fail for RE_status_result" );
832 2         status = COMMAND_RESULT_FAILURE;    /* fatal error */
834 1     }
834 1     else
835 2     {
836 2         out_args->status = RSTSL_SetMostRecentBackup( in_args->flags );
837 2         *output_args = (void *) out_args;
838 1     }
840 1     xdr_free( xdr_RE_set_backup_time_args, (char *)in_args );
841 1     free( in_args );
844 1     return status;
      }
```

```c
1   /*
2   ** ===================================================================
3   **
4   ** Copyright 1996, 1997 EMC Corporation
5   **
6   ** ===================================================================
7   */
8   /*
9   ** ===================================================================
10  **
11  ** EDMFinalStatus.c
12  **
13  ** ===================================================================
14  **
15  ** Mission Statement:
16  **
17  ** Primary Data Acted On:
18  **
19  ** Compile-Time Options:
20  **
21  **     USE_SUNRPC  - Compile source with sunrpc
22  **                   support.  If
23  **                   not set, assume DCE support.
24  **
25  ** Basic idea here: Module for Control Channel Reader thread.
26  **
27  ** The following provides an RCS id in the binary that can be located
28  ** with the what utility. The intent is to keep this short.
29  **
30  */
31  #if (defined(lint))
32  static char    RCS_id [] = "@(#)SRCSfile: EDMFinalStatus.c,v $ "
33                            "$Revision: 1.23 $ "
34                            "$Date: 1997/02/06 20:49:15 $";
35  #endif
36
37  /* #define  POSIX_SOURCE          unable to compile with this #define set */
38  /* #define  _XOPEN_SOURCE         unable to compile with this #define set */
39
40  // Rogue Wave includes
41  #include <sys/types.h>
42  #include <sys/utsname.h>
43  #include <sys/socket.h>
44  #include <netinet/in.h>
45  #include <netdb.h>
46  #include <arpa/inet.h>
47  #include <rw/c_portable.h>
48  #include <rw/ep_xopen.h>
49  #include <rw/inout.h>
50
51  #include <csc/cscomm.h>
52  #include <pthread.h>
53
54  #include <rw/cstring.h>
55  #include <rw/collect.h>
56  #include <rw/vref.h>
57  #include <rw/vstream.h>
58  #include <rw/bintree.h>
```

```c
60  #ifdef  __cplusplus
61  extern "C" {
62  #endif
63
64  #include <DDutils.h>
65  #include <restore/dispatch/Dispatch_Protocol_Client.h>
66  #include <restore/csc_Dispatch_Protocol_Client.h>
67  #include <restore/dispatch/dispatch_service.h>
68  #include <restore/dispatch_protocol.h>
69  #include <EDMFinalStatus.h>
70  #include <restore/dispatch/dispatch_protocol.h>
71
72  #ifdef  __cplusplus
73  }
74  #endif
75  #include <EDMRE_ccr.h>
76  #include <EDMSession.h>
77  #include <logging/logging.h>
78  #include <EDMReturnMessage.h>
79  #include <EDMReturnMessageApi.h>
80  #include <EDMRestoreEcdrog.h>
81
82  // Global/Extern if. spec to be used by RE_ccw/DD_ccr.
83  rpc_binding_handle_t *restoreService_ccw_handle_P;
84  extern DD_Client_session_id  *p_restoreServiceId;
85
86  int
87  SendFinalStatus(void)
88  {
89      int lrc=0;
90      int status=0;
91
92      lrc = PushResponseMessage( (int) dp_final_stats_indicate,
93                                 restoreService_ccw_handle_P,
94                                 restoreServiceId,
95                                 restoreService_ccw_handle_P,
96                                 &status );
97
98      return(lrc);
99  }
```

```c
 2
 4   /**********************************************************
 5   **
 6   ** File Name:    RSLinit.c
 7   **
 8   ** Copyright (c) 1998,1999 by EMC Corporation.
 9   **
10   ** Purpose:      This module contains the Restore Service Library
11   **               functions to
12   **               initialize and terminate the restore operation.
13   **
14   ** Table of Contents:
15   **
16   **               RSLInitialize
17   **               RSLFinish
18   **
19   **               Internal Functions:
20   **
21   ** Compile-Time Options:
22   **               This section must list any compile time definitions
23   **               which will affect this header.
24   **
25   ***********************************************************/

27   /* The following provides an RCS id in the binary that can be located
28      with the what(1) utility. The intent is to keep this short.
29   */
31   #ifndef lint
32   static char RCS_id [] = "$RCSfiles$ "
33                           "$Revision$ "
34                           "$Date$";
35   #endif

38   /*
39   ** Feature test switches
40   ** Standard defines required to turn on OS features go here.
41   ** The following is required for code that uses POSIX API's.
42   ** Remove for non-POSIX, non-portable code.
43   */
44
46   #define _POSIX_SOURCE 1

49   /*
50   ** System headers.
51   */
52   #include <sys/param.h>
53   #include <dirent.h>
54   #include <client.h>

57   /*
58   ** Epoch headers.
59   */
60   #include <eb/eb_port.h>
61   #include <eb/rb_log.h>
```

```c
64
65   /*
66   ** Local headers
67   */
68   #include <RSLintern.h>

71   #include <RSLstartup.h>

73   /*
74   ** #defines, structures, typedefs local to this source file
75   */
76   static int validate_plugin( restore_context *rcp );

79   /*
80   ** This is the global "restore context" that will be used
81   ** throughout the rest of the restore operations.
82   */
83   static restore_context *rcp = NULL;

85   /*
86   ** External declarations
87   */

89   /*
90   ** Definitions of the names of the plugin functions in the pifuncArray
91   ** of the pluginData structure.
92   **
93   ** as the PlFuncIndex values must be in the same order and position
94   ** of the plugindata values defined in RSLplugin.h
95   */
96   char *piFuncNames[PlFuncIndexLast+1] = {
97       "RSTPI_Identify",
98       "RSTPI_Initialize",
99       "RSTPI_GetToplevelObjects",
100      "RSTPI_GetNextToplevelObjects",
101      "RSTPI_GetNextObjects",
102      "RSTPI_GetRestoreContext",
103      "RSTPI_GetToplevelTemplates",
104      "RSTPI_DoesAlternateExist",
105      "RSTPI_MarkObject",
106      "RSTPI_IsObjectMarkable",
107      "RSTPI_GetAllBackupItems",
108      "RSTPI_IsObjectMarked",
109      "RSTPI_GetCurrentBackupTime",
110      "RSTPI_Submit",
111      "RSTPI_SetBackupTime",
112      "RSTPI_SetPrevBackup",
113      "RSTPI_SetNextBackup",
114      "RSTPI_SetFirstBackup",
115      "RSTPI_SetMostRecentBackup",
116      "RSTPI_IsTherePrevBackup",
117      "RSTPI_IsThereNextBackup",
118      "RSTPI_IsThereNextBackupForTime",
119      "RSTPI_IsTherePrevBackupForTime",
120      "RSTPI_Finish",
121      "RSTPI_StartRestore",
122      "RSTPI_FindRestorableObjects",
123      "RSTPI_GetFindResults",
124      "RSTPI_GetNecessaryMedia"
     };
```

```
127  /*****************************************************
128   * RSTSL_Initialize:
129   *
130   * This function takes care of all the initialization for a restore
131   * session. This must be called prior to any of the other functions
132   * in the restore API.
133   *
134   * Parameters:
135   *
136   *   userName (I)  - The name of the user.
137   *
138   *   now.
139   *
140   *****************************************************/
141
142  eerrno_ty
143  RSTSL_Initialize( const char *userName )
144  {
145    eerrno_ty status = E_SUCCESS;
146
147    /* If we have not yet allocated space for a restore_context
148     * structure, do so now. If we have already done so, just clear it
149     * now.
150     */
151    if ( NULL == rcp )
152    {
153      rcp = (struct restore_context *)malloc(sizeof(
                                      struct restore_context));
154      if ( NULL == rcp )
155      {
156        rec_api_log_csm(SUB_CSM_NOMEM, NULL);
157        return(EP_RB_RECOVER_NOMEM);
158      }
159    }
160    memset(rcp, 0, sizeof(struct restore_context));
161
162    rcp->gui_mode = 1;
163
164    /*
165     * Set the appropriate field in the recovery context to indicate
166     * that this recovery session is based on the Recover API.
167     * This flag is in place for historical reasons but is used by
168     * other parts of the Recover API library.
169     */
170    rcp->rec_human_uidname = esl_strdup( userName );
171
172    rcp->gui_mode = 1;
173
174    /*
175     * Initialize the logging mechanism.
176     */
177    if (status = rbrlog_begin(rcp, progname))
178    {
179      return(status);
180    }
181
182    /*
183     * Initialize the few 'recover context' variables that we can at
184     * this early stage.
185     */
```

```
191    setup_proc(rcp);
192
193    /*
194     * The following call will:
195     *   -Initialize the savsest databases
196     *   -Infer any information we can at this point.
197     */
198    status = startup(rcp);
199
201    if (status = startup(rcp))
202    {
203      return(status);
204    }
205
206    /* Do plugins setup: Find and initialize all valid restore plugin
207                                                             libs. */
208    status = init_plugins( rcp );
209
210    return status ;
211  } /* End of RSTSL_Initialize() */
```

```c
/*
 * *****************************************
 *
 * RSTBL_Finish
 *
 *   Function Description:
 *
 *   This function terminates a restoral session.
 *   It will be rejected if a restore is in
 *   progress.
 *   This routine will clean up any local memory used in the session.
 *
 *   Parameters:
 *
 *   none
 *
 * *****************************************
 */
errno_by
RSTBL_Finish( void )
{
    int mc_n;

    if (NULL == rcp)
    {
        return( E_SUCCESS );
    }
    RemoveSubmitFiles();

    /*
     * Call rbr_cleanup() which kills the aux proc(s), unlocks the work
     * item, then calls rbrlog_end() to enter the last logs and to close
     * the log file.
     */
    rbr_cleanup(rcp);

    /*
     * Deallocate the memory of restore_context and the related
     * structures.
     */
    if (NULL != rcp->rc_jmcp)
    {
        mcat_destroy(rcp->rc_jmcp);     /* Free the multicat structures */

        free(rcp->rc_jmcp);
        rcp->rc_jmcp = NULL;
    }

    /*
     * Free the mark bit map space
     */
    for (mc_n = 0; mc_n < rcp->rc_marks_plane_alloc; mc_n++)
    {
        if (NULL != rcp->rc_marks[mc_n])
        {
            free(rcp->rc_marks[mc_n]);
        }
    }
```

```c
        free(rcp->rc_marks_by_plane);
    }

    /*
     * Free the configuration structures
     */
#if 0
    if (NULL != rcp->rc_cfgname)
    {
        free(rcp->rc_cfgname);
    }
#endif

    if (NULL != rcp->rc_config)
    {
        rbc_freeconfig(rcp->rc_config);
    }

    /*
     * Free the DS_NONE structures array
     * Note that even though rc_dsnone is the head of linked list
     * of dsnone_info structures, the list is allocated via malloc
     * es an array initially (ref. alloc_plane_array()), therefore
     * we can do a free here.
     */
    if (NULL != rcp->rc_dsnone)
    {
        free(rcp->rc_dsnone);
    }

    /*
     * Free the volume list structures
     */
    if (NULL != rcp->rc_evblist)
    {
        (void)ebvl_volidlist_destructor(
            rcp->rc_evblist, EBVL_DESTROY_ALL);
    }

    /*
     * Free the plugin related data
     */
    rcp->rc_backup_app = 0;
    while (rcp->rc_currentPiptr = rcp->rc_pilist)
    {
        rcp->rc_backup_app++;
        rcp->rc_pilist = rcp->rc_currentPiptr->appdata;
        /* allow plugin to clean up and close */
        if ( E_SUCCESS != (err =
            rcp->rc_currentPiptr->pIFuncArray[ pIFuncIndexFinish ] (
            rcp) )
        {
            /* log error, continue */
            rbc_user_error( err,
                "RSTPL_Finish failed for restore plug-in
                library %s\n",
                ((struct pluginIDData *) rcp->
                rc_currentPiptr->libHd1 )->
                idData )-> name );
        }
        dlclose( rcp->rc_currentPiptr->libHd1 );
        rcp->rc_pilist = rcp->rc_currentPiptr->next;
    }
```

```
336 2      free (rcp->current_Pipt);
337 1

339 1      /*
340 1       * Free the various simple string buffers
341 1       */

343 1      if (NULL != rcp->rc_top_level_object_name)
344 1      {
344 2          free(rcp->rc_top_level_object_name);
346 1      }

348 1      if (NULL != rcp->rc_template_name)
349 2      {
350 2          free(rcp->rc_template_name);
351 1      }

353 1      if (NULL != rcp->rc_workitem_name)
354 2      {
355 2          free(rcp->rc_workitem_name);
356 1      }

358 1      if (NULL != rcp->rc_human_uidname)
359 2      {
360 2          free(rcp->rc_human_uidname);
361 1      }

363 1
364 2          /* don't free, its internal: free(rcp->rc_effective_uidname);
365 2           */
366 1      }

368 1      if (NULL != rcp->rc_client_rhuname)
369 2      {
370 2          free(rcp->rc_client_rhuname);
371 1      }

373 1      if (NULL != rcp->rc_client_hostname)
374 2      {
375 2          free(rcp->rc_client_hostname);
376 1      }

378 1      if (NULL != rcp->rc_client_scriptname)
379 2      {
380 2          free(rcp->rc_client_scriptname);
381 1      }

383 1      if (NULL != rcp->rc_cmd_context)
384 2      {
385 2          /* don't free -- its internal/temp data: free(
                  rcp->rc_cmd_context); */
386 1      }

388 1      if (NULL != rcp->rc_client_dirtop)
389 2      {
390 2          free(rcp->rc_client_dirtop);
391 1      }

393 1      if (NULL != rcp->rc_source_client_hostname)
394 2      {
394 2          free(rcp->rc_source_client_hostname);
396 1      }

396 1
398 1      if (NULL != rcp->rc_cpiogen_executable)
```

```
399 1      {
400 2          /* don't free, its internal: free(rcp->rc_cpiogen_executable); */
401 1      }

403 1      if (NULL != rcp->rc_plugin_wl_types)
404 2      {
405 2          free(rcp->rc_plugin_wl_types);
406 1      }

408 1      if (NULL != rcp->rc_pwd)
409 2      {
410 2          free(rcp->rc_pwd);
411 1      }

413 1      /*
414 1       * Finally, deallocate the restore_context itself
415 1       */

417 1      free(rcp);
418 1      rcp = NULL;

421        return( err );
422 1  }   /* RSTSL_Finish */
```

```
/*****************************************************************
 *
 *  init_plugins
 *
 *****************************************************************
 *
 *  Function Description:
 *
 *    This function locates, opens, validates and initializes all
 *    plug-in (shared) libraries. They must be located in
 *    /usr/epoch/EBI/cure_plugin (eb_cure_plugin_dir). All .so files in that
 *    directory are opened and validates for version and presence of all
 *    mandatory functions. The RSTPI_Identify function is called for each
 *    library to determine which optional features are supported and that
 *    the corresponding functions are present. Finally, the RSTPI_Initialize
 *    function is called for each valid library.
 *
 *  Parameters:
 *
 *    Inputs:
 *      rcp          (I)    - Pointer to restore context
 *
 *    Outputs:
 *      none
 *
 *  Returns:
 *      E_SUCCESS or EP_RB_RECOVER_xxx
 *
 *  Logic/pseudo code:
 *
 *      open plugin dir
 *      while read_next_entry succeeds
 *          while valid_types         (else continue)
 *          open shared library file
 *              on errors below:
 *                  close shared library file
 *                  continue
 *          fetch all mandatory function addresses
 *          call Identify function
 *          validate version number
 *          fetch all indicated optional function addrs
 *          call Initialize function
 *          add mandatory to composite exclusion list
 *          add to valid plugin list
 *
 *      close plugin dir
 *
 *****************************************************************/

static eerno_ty init_plugins( restore_context *rcp )
{
    DIR               *dirp;
    struct dirent     *dirent;
    eerno_ty          status = E_SUCCESS;
    struct plugindata *pluginDataPtr = NULL;
    struct plugindata *pliserv = NULL;
    int               val_result;
    struct plugindata *idPdataPtr;
    char              shlib_dirlen;
    int               shlib_dirlen;
    char              shlib_path [MAXPATHLEN];
```

```
    /* open plugin directory or bust */
    if ( NULL == (dirp = opendir( eb_cure_plugin_dir )))
    {
        rec_api_log_can( SUB_CSM_PLUGIN_ERR, NULL );

        return E_SUCCESS;             /* allow continuation w/o plugins */
    }
#if 1
#else

        return RP_RB_RECOVER_NOMEM;    /* later do this */

    }
#endif

    /* loop thru entries in directory */
    while (NULL != (dirent = readdir( dirp )))
    {
        if (NULL == strstr( dirent->d_name, ".so" ))
            continue;               /* fail thru to cleanup */

        /* allocate next plugin data structure */
        if (NULL == (pluginDataPtr =
                (plugindata *)libhnd =
                calloc( 1, sizeof(
                struct plugindata) )))
        {
            status = RP_RB_RECOVER_NOMEM;
            break;
        }

        strcpy( shlib_path, eb_cure_plugin_dir );
        strcat( shlib_path, "/" );
        shlib_dirlen = strlen (shlib_path);

        strcpy( &shlib_path[shlib_dirlen], dirent->d_name );
        if (NULL == (pliserv->libhnd =
                dlopen( shlib_path, RTLD_NOW )))
        {
            the_user_error( 0,
                "Error opening restore plug-in library %s: %s\n",
                dirent->d_name, dlerror() );
            continue;               /* skip this one */
        }

        /* Fetch addresses of all mandatory functions and */
        /* Do Identify processing: call it, save options, validate */

        if (0 != (val_result = validate_plugin( pluginDataPtr )))
        {
            if (val_result == -1 || val_result == -4)
            {
                the_user_error( 0,
                "functions missing from restore plug-in library %s: %s\n",
                    dirent->d_name, dlerror() );
            }
            else if (val_result < 0)
            {
                the_user_error( 0,
                    "Validation failed for restore plug-in library %s\n",
                    dirent->d_name );
```

```c
        else
          {
            rbe_user_error( val_result,
              idDataPtr->pi1_types,
              "RSTPI_Identify failed for restore plug-in library
              %s\n",

              direntry->d_name );

            dlclose( pl1DataPtr->libHdl );    /* close .so on errors */
            continue;    /* on any error, skip this lib */
          }
      }

/* let DC plug-in do its initialization */
      rcp->appData = NULL;
      pl1DataPtr->piDataPtr = NULL;   /* enter plugin with clean appdata */
      status =
        pl1DataPtr->piFuncIndexInitialize]( rcp );
      if (E_SUCCESS != status)
      {
        rbe_user_error( status,
          "RSTPI_Initialize failed for restore plug-in library
          %s\n",
            direntry->d_name );

          dlclose( pl1DataPtr->libHdl );    /* close .so on errors */
          pl1DataPtr->libHdl = NULL;
          status = E_SUCCESS;    /* this wasn't fatal */

          continue;    /* on any error, skip this lib */
      }

/* add pl1DataPtr to valid plugin list */
      if (NULL == pl1istPtr)    /* first in list */
      {
        rcp->pl1ist = pl1DataPtr;
      }
      else
      {
        pl1istPtr->next = pl1DataPtr;    /* link from prev */
      }
      pl1istPtr = pl1DataPtr;    /* new end of list */

/* move old list to new buffer and free old list */
      if ( NULL != rcp->rc_plugin_wl_types)
      {
        tmp_types = calloc( 1, 1 + idDataPtr->num_types
                  + rcp->rc_num_plugin_wl_types
                  );
        if (NULL == tmp_types) {
          status = EP_RB_RECOVER_NOMEM;
          break;
        }
        memcpy( tmp_types,
          rcp->rc_plugin_wl_types,
          rcp->rc_num_plugin_wl_types );
```

```c
        memcpy( tmp_types + rcp->rc_num_plugin_wl_types,
          idDataPtr->pi1_types,
          idDataPtr->num_types );

        free( rcp->rc_plugin_wl_types );
        rcp->rc_num_plugin_wl_types += idDataPtr->num_types;
        rcp->rc_plugin_wl_types = tmp_types;

        tmp_types[rcp->rc_num_plugin_wl_types] = 0;
        rcp->rc_plugin_wl_types = tmp_types;
      }
    }

    (void)closedir( dirp );

/* free up leftovers: */
    if (NULL != pl1DataPtr)
      free (pl1DataPtr);

/* Free contents of plugin list: */
    if (E_SUCCESS != status)
    {
      /* Free plugin .so to clean up and close: */
      while (NULL != (pl1DataPtr = pl1istPtr))
      {
        /* allow plugins to clean up and .so: */
        rcp->appData = pl1DataPtr->appData;
        pl1DataPtr->piFuncIndexFinish]( rcp );

        dlclose( pl1DataPtr->libHdl );
        pl1istPtr = pl1DataPtr->next;
        free (pl1DataPtr);
      }
    }

    return status;
}
```

```
626            /* init_plugins */
628
629
630     2    /********************************************************
631     2     * validate_plugin
632     1     *
633     1     * Function Description:
634           *
635           * This function retrieves the addresses of the mandatory plugin
636     1     *                                                functions
637           * and stores the function pointer array.
638     2     *
639           * It then calls the identify function and verifies whbe plugin
640     1     *
641           * version. It finds its optional functions. Specific error values are
642           * returned on version mismatch and missing optional functions
643           *
644           * if it returns then the function is missing
645           *
646           * Parameters:
647           *
648     2     *   pidataPtr    I
649     2     *
650     1     *        - pointer to plugin data structure with libHdl set
651           *
652           * Inputs:
653           *
654           *   pidataPtr  is loaded with pointers to plugin
655           *                                                functions
656           * Outputs:
657           *
658           * piFuncArray in pidataPtr is loaded with pointers to plugin
659           *                                                functions
660     1     * Returns:
661     1     *
662     2     *   0 on success
663           *
664     1     *   -1 on any missing required functions
665     2     *
666     2     *   -2 if version validation fails OR identify returns junk
667     2     *
668     2     *   -3 if version mismatch and missing optional functions
669           *
670     2     *   -4 on any missing optional functions indicated by options
671     1     *                                                flags
672     1     *   +n
673     1     *
674     1     * EB_RB_RECOVER_xxxx for error codes returned from identify function
675     1     *
676     1     ********************************************************/
677
678     1    static int validate_plugin( struct pluginData *piData )
679          {
680               int                index;
681               status_ty          status;
682               struct pluginData  *idPtr = &idData );
683
684               for ( index = 0; index <= PiFuncIndexLastBasic; index++ )
685               {
686                   if ( NULL == (pidataPtr->piFuncArray[index]
687                                  = (piFuncPtr) dlsym( pidataPtr->libHdl,
688                                        piFuncNames[index]
689
690                   {
691                       return -1;
692                   }
693               }
694
695               if (status != E_SUCCESS)
696                   return status;
697               if (NULL == (idataPtr->idData )
698                   return -2;
699                   struct pluginData *)pidataPtr->idData )
```

```
680     1        if (idataPtr->version != RSTPL_VERSION )
681     2        {  /* We only support version 1 supported so for */
682     2            pidataPtr->idData = NULL;
683     2            return -2;
684     1        }
685
686     1        /* if startRestore option set, get its addr or bust */
687     2        if ( ( (idataPtr->num_types && (idataPtr->vi_types)
688     2             == (RSTPL_OPTION_SPECIAL_START & RSTPL_OPTION_MASK_START)
689     2             && (NULL ==
690     1                  (idataPtr->piFuncArray[PiFuncIndexStartRestore]
691                       = (piFuncPtr) dlsym( pidataPtr->libHdl,
692                           piFuncNames[PiFuncIndexStartRestore] ) ) ) )
693
694     1            /* count cant be positive with null pointer */
695     1            pidataPtr->options = pidataPtr->options & RSTPL_OPTION_MASK_START;
696
697
698     1        /* if special find option set, get its addr or bust */
699     2        if ( ( (idataPtr->options & RSTPL_OPTION_SPECIAL_FIND)
700     2             == (RSTPL_OPTION_SPECIAL_FIND & RSTPL_OPTION_MASK_FIND)
701     2             && (NULL ==
702     1                  (idataPtr->piFuncArray[PiFuncIndexFind]
703                       = (piFuncPtr) dlsym( pidataPtr->libHdl,
704                           piFuncNames[PiFuncIndexFind] ) ) ) )
705     1            || (NULL ==
706                  (idataPtr->piFuncArray[PiFuncIndexFindResults]
707                   = (piFuncPtr) dlsym( pidataPtr->libHdl,
708                       piFuncNames[PiFuncIndexFindResults] ) ) )
709
710     2            pidataPtr->options = pidataPtr->options & RSTPL_OPTION_MASK_FIND;
711     2            return -4;
712     1        }
713
714     1        /* if special getMedia option set, get its addr or bust */
715     2        if ( ( (idataPtr->options & RSTPL_OPTION_GET_MEDIA)
716     2             == (RSTPL_OPTION_SPECIAL_GET_MEDIA & RSTPL_OPTION_MASK_GET_MEDIA)
717     1             && (NULL ==
718                      (idataPtr->piFuncArray[PiFuncIndexGetMedia]
719                       = (piFuncPtr) dlsym( pidataPtr->libHdl,
720                           piFuncNames[PiFuncIndexGetMedia] ) ) ) )
721     1            pidataPtr->idData = NULL;
722     1            return -4;
723          }
724
725               return 0;
726          }
```

723

```
/*
                   validate_plugin */
```

```
1   /*!
2   ** Copyright 1996,1997 EMC Corporation
3   **
    ** EDMReturnMessageApi.cc
6   **
7   ** Mission Statement:  file that contains an API to manage the Message
8   **                                                                Queues
9   **
10  ** Primary Data Acted On:
11  **
12  ** Compile-Time Options:
13  **
14  ** Basic Idea Here:
15  **                 A few calls to manage the Message Queues.
16  **
17  **
    */

19  #if !defined(lint)
20  static char     RCS_id [] = "@(
                    #$RCSfile: EDMReturnMessageApi.cc,v $ "
                    "$Revision: 1.0 $ "
                    "$Date: 1997/02/06 20:49:15 $ ";
    #endif

23  #include <self/c_portable.h>
25  #include <self/ep_kopen.h>
26  #include <self/inout.h>
27

29  #include <stdlib.h>
30  #include <sys/types.h>
31  #include <pthread.h>

33  #include <logging/logging.h>
34  #include <csr/csrcomm.h>
35  #include <errno/e_eb.h>

38  // Rogue Wave includes
39  #include <rw/collect.h>
40  #include <rw/rwfile.h>
41  #include <rw/vstream.h>
42  #include <rw/gqueue.h>

44  #ifdef __cplusplus
45  extern "C" {
46  #endif

48  #include <restore/dispatch_daemon.h>

50  #ifdef __cplusplus
51  }
52  #endif

54  #include <EDMSession.h>
55  #include <EDMHandlingMgr-Api.h>
56  #include <EDMReturnMessage.h>
57  #include <EDMReturnMessageApi.h>

59  declare(RWQueue, EDMReturnMessage)

61  RWGQueue(EDMReturnMessage) g_messageQueue;

63  static pthread_mutex_t g_returnmessagemutex =
                           PTHREAD_MUTEX_INITIALIZER;
```

```
65
66  /*****************************************************************
67  **   Routine:    LockReturnMessageMutex
68  **
69  **   Inputs:     None
70  **
71  **   Outputs:    None
72  **
73  **   Return Codes:
74  **               None
75  **
76  **   Purpose:    Lock the mutex for the return message object
77  **
78  **
    */
79

81  static void
    LockReturnMessageMutex()
    {
84 1    static boolean_ty first = TRUE;

86 1    if (first == TRUE)
87 2    {
88 2        first = FALSE;
89 2        pthread_mutex_init(&G_returnmessagemutex, NULL);
90 1    }

92 1    pthread_mutex_lock(&G_returnmessagemutex);
93 1 }
95
96  /*****************************************************************
97  **   Routine:    UnlockReturnMessageMutex
98  **
99  **   Inputs:     None
100 **
101 **   Outputs:    None
102 **
103 **   Return Codes:
104 **               None
105 **
106 **   Purpose:    Unlock the mutex for the return message object
107 **
108 **
    */
109

111 static void
    UnlockReturnMessageMutex()
112 {
113 1    pthread_mutex_unlock(&G_returnmessagemutex);
114 1 }
115

117 /*****************************************************************
118 **   Routine:    PushResponseMessage
119 **
120 **   Inputs:     int msgtoqueue  -- an integer representing the message to
121 **                                                  queue.
122 **               DD_client_session_id sid  -- Source service of message.
123 **
```

```
124  ** Outputs:     int  *status - a status for the push command
125  **
126  ** Return Codes:
127  **              0 for success and non-zero for failure.
128  **
129  ** Purpose:    push a message on the Message Queue.
130  **
131  **
132  **************************************************************************/
133
134  int
136  PushResponseMessage(IN int msgtoqueue,
137                      IN DD_client_session_id sid,
138                      IN rpc_binding_handle_t *csc_h,
139                      OUT int *status)
140  {
141      CSC_handle_t *csc_h;
142
143      // At one point we did check the csc_h handle) but we found that
144      // we would rather take a NULL and check it once we try to use it.
145      // That way we can handle cases where the private service messages
146      //   us
148      // and we haven't made a connection to it yet.
149
150      if (msgtoqueue == 0)
151      {
152          *status = RETURNMESSAGE_BAD_PARAM;
153          return 1;
154      }
155
156      EDMReturnMessage *msg, *ret;
158
159      if (status == NULL)
160      {
161          return 1;
162      }
163
164      // Acquire a new message Object:
165
166      msg = new EDMReturnMessage();
168      if (msg == NULL)
169      {
170          *status = RETURNMESSAGE_NO_MEMORY;
171          return(1);
172      }
173
174      // Build the message
175
176      msg -> setSessionID(sid);
        msg -> setMessage(msgtoqueue);
        msg -> setTimeIssued(time(NULL));
        msg -> setBindingHandle(csc_h);
178      LockReturnMessageMutex();

        ret = g_messageQueue.append(msg);

        UnlockReturnMessageMutex();

        if (ret = NULL)
        {
            *status = RETURNMESSAGE_QUEUE_APPEND_FAILURE;
            delete msg;
            return 1;
        }
```

```
187  }
188
189      return 0;
190  }
191
192
193  /**************************************************************************
194  **
195  ** Routine:   PopResponseMessage
196  **
197  ** Inputs:
198  **
199  ** Outputs:    int  *status - a status for the pop message
200  **             int  *msgid   - a place to put the message to queue
201  **             DD_client_session_id *sess - a place to put the
202  **                 session                       ID
203  **             rpc_binding_handle_t *client_h - a handle to respond
204  **                                               on.
205  **
206  ** Return Codes:
207  **              0 for success and non-zero for failure.
208  **
209  ** Purpose:   Pop a message off the Message Queue.  If the Queue is empty
210  **            block until a signal wakes us up.  If the timeout value is
211  **            0 the thread blocks until a message is received, otherwise
212  **            it will block the length of time specified.
213  **
214  **
215  **************************************************************************/
216  */
217
218  int
219  PopResponseMessage(
220                     OUT *ResponseMessage, OUT DD_client_session_id *sess,
221                     OUT int *msgid,
222                     OUT rpc_binding_handle_t *client_h,
223                     OUT int *status)
224  {
225
226      *status = 0;
228
229      if (client_h == NULL || ResponseMessage == NULL || sess == NULL)
230      {
231          *status = RETURNMESSAGE_BAD_PARAM;
232          return 1;
233      }
234
236      LockReturnMessageMutex();
237
238      if (g_messageQueue.isEmpty())
239      {
240          *status = RETURNMESSAGE_QUEUE_IS_EMPTY;
241          UnlockReturnMessageMutex();
242          return 1;
243      }
245      ret = g_messageQueue.get();
         UnlockReturnMessageMutex();
```

```
247  1      if (ret == NULL)
248  2      {
249  2          *status = RETURNMESSAGE_RECORD_GET_FAILED;
250  2          return -1;
251  1      }

253  1      ret -> getSessionID(sess);
254  1      *ResponseMessage = ret -> getMessage();
255  1      *client_h_p = ret -> getBindingHandle();

257  1      delete ret;

259  1      return 0;
260  1  }
```

EDMReturnMessageApi.cc 7

EDMReturnMessageApi.cc 8

```c
/***********************************************************************
**
** File Name:      EDMDispProtocolSvc.c
**
** Copyright (c) 1998,1999 by EMC Corporation.
**
** Purpose:
**     This module contains the callback functions for use with
**     the dispatch daemon protocol.
**
** Table of Contents:
**
** Compile-Time Options:
**     This section must list any compile time definitions
**     which will affect this header.
**
** Copyright (c) 1996 by EMC Corp.
**
***********************************************************************/

#if !defined(lint)
static char RCS_id[] = "@(#)$RCSfile: EDMDispProtocolSvc.c,v $ "
                       "$Revision: 1.0 $ "
                       "$State: 1999/03/06 09:00:00 $ ";
#endif

#include <esl/c_portable.h>
#include <esl/inout.h>
#include <util/esl_string.h>

#include <logging/logging.h>
#include <pthread.h>
#include <sys/time.h>
#include <errno.h>
#include <serrno/e_eb.h>
#include <EDMutile.h>

#ifdef __cplusplus
extern "C" {
#endif

#include <restore/dispatch_daemon.h>
#include <restore/dispatch_protocol.h>
#include <restore/dispatch_protocol_service.h>
#include <restore/dispatch_protocol_client.h>
#include <restore/EDMDD_ddp.h>
#include <restore/csc_dispatch_Protocol_Service.h>

#include <EDMDispatchSession.h>
#include <EDMTimeMessageApi.h>
#include <EDMReturnMessageApi.h>
#include <EDMDDlandlaWgrApi.h>

#include <logging/logging.h>
#include <csc/oscomm.h>
#include <EDMDispatchLog.h>

#ifdef __cplusplus
}
#endif
```

```c
/***********************************************************************
**
**                              Structures
**
***********************************************************************/

/***********************************************************************
**
** Routine:     dp_connect_indicate_1()
**
** Inputs:      None
**
** Outputs:     None
**
** Return Codes:
**              None
**
** Purpose:
**
** Intended caller:   Internal Only.
**
***********************************************************************/
int *
dp_connect_indicate_1_svc(
        Dp_connect_indicate_msg *msg, struct svc_req *req)
{
    int rc;
    int status;
    rpc_binding_handle_t client_handle_p = NULL;

    /* Update last time we heard from the service */
    rc = UpdateSessionLastReceived( &msg->sid );
    if (0 != rc)
    {
        EDMDispatch_logent(
            __FILE__, __LINE__, LOG_ERR, DDP_UPDATE_SESSION_RCV_FAILURE, 0,
            "UpdateSessionLastReceived failed.");
    }

    /* Get the csc_binding_handle associated with this sid */
    rc = GetCSCHandle(&msg->sid,
                      &client_handle_p,
                      &status );
    if (0 != rc)
    {
        EDMDispatch_logent(
            __FILE__, __LINE__, LOG_ERR, DDP_GET_CSC_HANDLE_FAILURE, status,
            "GetCSCHandle failed.");
    }

    /* Push message to send onto the queue */
    rc = PushResponseMessage((int) dp_connect_confirm,
                             &msg->sid,
                             client_handle_p,
                             &status);
    if (0 != rc)
    {
        EDMDispatch_logent(
            __FILE__, __LINE__, DDP_PUT_RESPONSE_FAILURE, status,
            "PushResponseMessage failed.");
    }
}
```

```
123 2    {
124 2       if (isDebugOn())
125 2       {
126 2          (void) EMDDispatch_logent(
                  __FILE__, __LINE__, LOG_ERR, DDP_SENDING_MESSAGE, 0,
                  "Pushing response message to restore service.");
127 1       }
129 1       return((int*)0);
130       }
```

```
132    /****************************************************************
133    **
134    ** Routine:  dp_abort_response_1()
135    **
136    ** Inputs:      None
137    **
138    ** Outputs:     None
139    **
140    ** Return Codes:
141    **              None
142    **
143    ** Purpose:
144    **              Intended caller:   Internal Only.
145    **
146    **
147    */
148
149    int *
150    dp_abort_response_1_svc( DP_abort_response_msg *msg, struct svc_req *req)
151    {
152       int rc;
          int status;

155       /* Update last time we heard from the service */
156       rc = UpdateSessionLastReceived( &msg->sid );
157       if (0 != rc)
158       {
159          EMDDispatch_logent(
                __FILE__, __LINE__, LOG_ERR, DDP_UPDATE_SESSION_RCV_FAILURE, 0,
                "UpdateSessionLastReceived failed.");
160       }
161
          /* Remove the timed message to indicate that we got the response */
163       rc = deleteTimedMessage(&msg->sid,
164                                dp_abort_request,
                                   &status);
165       if (0 != rc)
166       {
167          EMDDispatch_logent(
                __FILE__, __LINE__, LOG_ERR, DDP_DELETE_TIMED_MSG_FAILURE, status,
                "deleteTimedMessage failed.");
169       }
170
171       return( (int*)0 );
174    }
```

```
176  /***************************************
177  **
178  ** Routine:   dp_close_response_1()
179  **
180  ** Inputs:    None
181  **
182  ** Outputs:   None
183  **
184  ** Return Codes:
185  **            None
186  **
187  ** Purpose:
188  **
189  ** Intended caller:  Internal Only.
190  **
191  ***************************************/

193  int *
194  dp_close_response_1_svc( DP_close_response_msg *msg, struct svc_req *req)
195  {
196      int rc;
197      int status;

199      /* Update last time we herd from the service */
200      rc = UpdateSessionLastReceived( &msg->sid );
201      if (0 != rc)
202      {
203          EDMDispatch_logent(
204              __FILE__, __LINE__, LOG_ERR, DDP_UPDATE_SESSION_RCV_FAILURE, 0,
                     "UpdateSessionLastReceived failed." );
206      }

207      /* Remove the timed message to indicate that we got the response */
208      rc = deleteTimedMessage(&msg->sid,
209                              dp_close_request,
210                              &status);
211      if (0 != rc)
212      {
213          EDMDispatch_logent(
214              __FILE__, __LINE__, LOG_ERR, DDP_DELETE_TIMED_MSG_FAILURE, status,
                     "deleteTimedMessage failed." );
215      }

217      return( (int*)0 );
218  }
```

```
219  /***************************************
220  **
221  ** Routine:   dp_ping_response_1()
222  **
223  ** Inputs:    None
224  **
225  ** Outputs:   None
226  **
227  ** Return Codes:
228  **            None
229  **
230  ** Purpose:
231  **
232  ** Intended caller:  Internal Only.
233  **
234  ***************************************/

236  int *
237  dp_ping_response_1_svc( DP_ping_response_msg *msg, struct svc_req *req)
238  {
239      int rc;
240      int status;

242      /* Update last time we herd from the service */
243      rc = UpdateSessionLastReceived( &msg->sid );
244      if (0 != rc)
245      {
246          EDMDispatch_logent(
247              __FILE__, __LINE__, LOG_ERR, DDP_UPDATE_SESSION_RCV_FAILURE, 0,
                     "UpdateSessionLastReceived failed." );
248      }

250      /* Remove the timed message to indicate that we got the response */
251      rc = deleteTimedMessage(&msg->sid,
252                              dp_ping_request,
253                              &status);
254      if (0 != rc)
255      {
256          EDMDispatch_logent(
257              __FILE__, __LINE__, LOG_ERR, DDP_DELETE_TIMED_MSG_FAILURE, status,
                     "deleteTimedMessage failed." );
258      }

260      return( (int*)0 );
261  }
```

```
263
264    /*****************************************
265    **
266    ** Routine:   dp_event_indicate_1()
267    **
268    ** Inputs:    None
269    **
270    ** Outputs:   None
271    **
272    ** Return Codes:
273    **
274    ** Purpose:
275    **
276    ** Intended caller:   Internal Only.
277    **
278    */
279
280    int *
281    dp_event_indicate_1_svc(
282                    DP_event_indicate_msg *msg, struct svc_req *req)
283    {
284        int    rc;
285        int    status;
286        rpc_binding_handle_t *client_handle_p;
287
288        /* Update last time we heard from the service */
289        rc = UpdateSessionLastReceived( &msg->sid );
290        if (0 != rc)
291        {
292            EDMDispatch_logent(
293                __FILE__, __LINE__, LOG_ERR, DDP_UPDATE_SESSION_RCV_FAILURE, 0,
294                "UpdateSessionLastReceived failed.");
295        }
296        return( (int*)0 );
```

```
297
298    /*****************************************
299    **
300    ** Routine:   dp_progress_indicate_1()
301    **
302    ** Inputs:    None
303    **
304    ** Outputs:   None
305    **
306    ** Return Codes:
307    **
308    ** Purpose:
309    **
310    ** Intended caller:   Internal Only.
311    **
312    */
313
314    int *
315    dp_progress_indicate_1_svc(
316                    DP_progress_indicate_msg *msg, struct svc_req *req)
317    {
318        int    rc;
319        int    status;
320
321        /* Update last time we heard from the service */
322        rc = UpdateSessionLastReceived( &msg->sid );
323        if (0 != rc)
324        {
325            EDMDispatch_logent(
326                __FILE__, __LINE__, LOG_ERR, DDP_UPDATE_SESSION_RCV_FAILURE, 0,
327                "UpdateSessionLastReceived failed.");
328        }
329        return( (int*)0 );
```

```
331
332  /**********************************************************
333  **
334  ** Routine:   dp_final_stats_indicate()
335  **
336  ** Inputs:    None
337  **
338  ** Outputs:   None
339  **
340  ** Return Codes:
341  ** None
342  **
343  ** Purpose:
344  **
345  ** Intended caller: Internal Only.
346  **
     ********************************************************/

348  int *
349  dp_final_stats_indicate_1_svc(DP_final_stats_indicate_msg *msg,
350                                  struct svc_req *req)
351  {
352  1  int rc;
353  1  int status;
354  1  rpc_binding_handle_t *client_handle_p = NULL;

356  1  /* Update last time we herd from the service */
357  1  rc = UpdateSessionLastReceived(&msg->sid );
358  1  if (0 != rc)
359  2  {
360  2      EDMDispatch_logent(
                __FILE__, __LINE__, LOG_ERR, DDP_UPDATE_SESSION_RCV_FAILURE, 0,
                "UpdateSessionLastReceived failed.");
361  2  }

362  1  /* Get the csc_binding_handle associated with this sid */
363  1  rc = GetCSCHandle(&msg->sid,
364  1                    &client_handle_p,
365  1                    &status);
366  1  if (0 != rc)
367  2  {
368  2      EDMDispatch_logent(
                __FILE__, __LINE__, LOG_ERR, DDP_GET_CSC_HANDLE_FAILURE, status,
                "GetCSCHandle failed.");
369  2  }
370  2

371  1

373  1  /* Push message to send onto the queue */
374  1  rc = PushResponseMessage(msg,
375  1                           msg->sid,
376  1                           client_handle_p,
377  1                           &status);
378  1  if (0 != rc)
379  2  {
380  2      EDMDispatch_logent(
                __FILE__, __LINE__, LOG_ERR, DDP_PUT_RESPONSE_FAILURE, status,
                "PushResponseMessage failed.");
381  2  }
382  1

384  1  return( (int*)0 );
385  }
```